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.

UMI
University Microfilms International
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
313/761-4700 800/521-0600
Object-oriented models for numerical and finite element analysis

Yu, George Gang, Ph.D.
The Ohio State University, 1994

Copyright ©1994 by Yu, George Gang. All rights reserved.
OBJECT-ORIENTED MODELS FOR
NUMERICAL AND FINITE ELEMENT ANALYSIS

DISSERTATION

Presented in Partial Fulfillment of the
Requirements for the Degree Doctor of Philosophy in
the Graduate School of The Ohio State University

By

George Gang Yu, B. E., M. E.

The Ohio State University

1994

Dissertation Committee:

Prof. Hojjat Adeli (Adviser)
Prof. Robert Sierakowski
Prof. Shive Chaturvedi
Prof. Mukesh Singhal

Approved by

Adviser
Department of Civil Engineering
To My Wife, Glenda
ACKNOWLEDGMENTS

I would like to acknowledge Dr. Hojjat Adeli as my Ph.D. advisor. I would also like to thank Dr. Robert Sierakowski and Dr. Shive Chaturvedi for my composite laminate study, Dr. Dik Lee for my computer science study, and Dr. Mukesh Singhal for serving on my Dissertation Committee. To my fellow students, I would especially like to thank Mr. Teddy Tzeng for his cooperation and support in our research project. Additionally, the computing facilities and supporting staff in the Department of Computer Science and College of Engineering CAD/CAM Laboratory are highly appreciated.

Finally, to my wife, Glenda, I offer my deepest love and appreciation. Without her understanding and support, I could not have achieved my goal.
VITA

September 17, 1960 .......................... Born, Shanghai, China

1983 ................................. B.E., Shanghai College of Architectural & Municipal Engineering, China

1988 ................................. M.E., The University of Tokyo, Japan

1992-Present .......................... Research Scientist, Battelle Memorial Institute, Columbus, OH
PUBLICATIONS


FIELDS OF STUDY

Major Field: Civil Engineering

  Structural Engineering: Dr. Hojjat Adeli
  Composite Laminates: Drs. Robert Sierakowski and Shive Chaturvedi

Minor Field:

  Computer Science: Dr. Dik Lee
TABLE OF CONTENTS

DEDICATION iii
ACKNOWLEDGMENTS iv
VITA v
LIST OF TABLES xi
LIST OF FIGURES xii

CHAPTER PAGE

I. INTRODUCTION ................................................................. 1
  1.1 General ................................................................. 1
  1.2 Related Work ......................................................... 1
  1.3 Objectives ............................................................ 2
  1.4 Organization of this Dissertation .............................. 4

II GLOBAL-LOCAL MODEL FOR COMPOSITE LAMINATE ANALYSIS 8
  2.1 Introduction ........................................................... 8
  2.2 The Global-Local Model ............................................ 10
  2.3 Discretization of Global-Local Model ......................... 11
    2.3.1 Discretization of Global Region ....................... 14
    2.3.2 Local Region ................................................ 20
  2.4 Assembly of Global And Local Region Stiffness Matrix 25

III. AN ARCHITECTURE OF OBJECT-ORIENTED PROGRAMMING IN COMPUTER-AIDED ENGINEERING .......... 27
  3.1 Introduction .......................................................... 27
  3.2 Blackboard Architecture ......................................... 28
  3.3 Principles of Blackboard Architecture for Numerical Analysis 31
  3.4 Implementation of Blackboard Architecture for Numerical Analysis 36
    3.4.1 Object-Oriented Programming Paradigm .......... 36
    3.4.2 Group of Blackboards ................................. 39
C. CLASS LIBRARY OF OBJECT-ORIENTED FINITE ELEMENT ANALYSIS ................................................................. 268
   C.1 Source Code of GENERAL.H .......................................................................................................................... 268
   C.2 Source Code of GENERAL.C .......................................................................................................................... 271
   C.3 Source Code of FEM.H ................................................................................................................................. 273
   C.4 Source Code of FEM.C .................................................................................................................................. 281

D. SOURCE CODE OF CONCURRENT OBJECT-ORIENTED COMPOSITE LAMINATE ANALYSIS ................................. 309
   D.1 Source Code of GL.H ...................................................................................................................................... 309
   D.2 Source Code of GL.C ...................................................................................................................................... 316
   D.3 Source Code of MAIN.C .................................................................................................................................. 342

E. DISSERTATION ABSTRACT ................................................................................................................................. 348

F. INDEX .................................................................................................................................................................. 350
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. A Survey of OOP Applications in Civil Engineering</td>
<td>3</td>
</tr>
<tr>
<td>2. Test results of solving systems of linear equations using concurrent object-oriented iterative methods</td>
<td>192</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. The global-local model</td>
<td>10</td>
</tr>
<tr>
<td>2. Stress and strain definition</td>
<td>13</td>
</tr>
<tr>
<td>3. Illustration of the interlaminar nodal displacements</td>
<td>17</td>
</tr>
<tr>
<td>4. Assembly of element stiffness matrix</td>
<td>26</td>
</tr>
<tr>
<td>5. Framework of blackboard architecture</td>
<td>29</td>
</tr>
<tr>
<td>6. General blackboard architecture for numerical analysis</td>
<td>32</td>
</tr>
<tr>
<td>7. Class representation of BANA method of the class</td>
<td>38</td>
</tr>
<tr>
<td>8. Group of blackboards for alternative design objects</td>
<td>40</td>
</tr>
<tr>
<td>9. A laminate</td>
<td>43</td>
</tr>
<tr>
<td>10. The global-local model</td>
<td>43</td>
</tr>
<tr>
<td>11. Element stiffness matrix computation controlling system</td>
<td>47</td>
</tr>
<tr>
<td>12. Program controlling system</td>
<td>48</td>
</tr>
<tr>
<td>13. Hierarchy of knowledge sources in Evaluation KS</td>
<td>50</td>
</tr>
<tr>
<td>14. A four-node element</td>
<td>54</td>
</tr>
<tr>
<td>15. Entity NODE in the ER model for general FEA</td>
<td>54</td>
</tr>
<tr>
<td>16. Entity NODE in the ER model for composite laminate analysis</td>
<td>55</td>
</tr>
<tr>
<td>17. Entity NODE_GL in the EER data model for composite laminate analysis</td>
<td>57</td>
</tr>
<tr>
<td>18. Class for Entity NODE in Figure 16</td>
<td>58</td>
</tr>
<tr>
<td>19. Class for Entity NODE_GL in Figure 17</td>
<td>58</td>
</tr>
</tbody>
</table>
46. File manager....................................................................................................105
47. 3+ index system for objects stored on the secondary storage......................106
48. Class card of class NODE with dynamic data management model ...............111
49. Object Version Tree ....................................................................................117
50. Paths to Versions ......................................................................................117
51. Relationships among data, class, object and version ...................................122
52. Parallel Version graph..................................................................................124
53. Data structure of object version manager .....................................................126
54. Class card of class Object_Record ...............................................................129
55. Class card of class Version_Record ..............................................................130
56. Class card of class OVM ..............................................................................131
57. A composite laminate under external force F.................................................132
58. Class relationships in formulating global region stiffness matrix ...................133
59. Element configuration for the global-local model .........................................134
60. Object and version records ........................................................................136
61. A general blackboard architecture for CAE applications ...............................143
62. Concurrent Execution ................................................................................147
63. Calling system function fork() ....................................................................148
64. Class card of class SEMAPHORE ..............................................................151
65. An example of creating a pipe ....................................................................152
66. Generating multiple processes and pipes .....................................................153
67. An example of message passing via a queue example ...................................155
68. Shared memory example ...........................................................................156
69. A composite laminated structure ................................................................157
70. Synchronization graph of task Input ..........................................................160
71. Synchronization graph of task Stiff ............................................................162
<table>
<thead>
<tr>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>72.</td>
<td>Synchronization graph of task E1</td>
</tr>
<tr>
<td>73.</td>
<td>Speedup</td>
</tr>
<tr>
<td>74.</td>
<td>Variations of user and system CPU times</td>
</tr>
<tr>
<td>75.</td>
<td>A finite element grid with N elements and ((1.5N+3)) nodes</td>
</tr>
<tr>
<td>76.</td>
<td>Class card of a full vector class \texttt{VECTOR}</td>
</tr>
<tr>
<td>77.</td>
<td>Class card of a sparse vector class \texttt{SPARSE_VECTOR}</td>
</tr>
<tr>
<td>78.</td>
<td>Comparison of memory requirement for element 1 in Figure 75, full vector approach vs. sparse vector approach</td>
</tr>
<tr>
<td>79.</td>
<td>Concurrent computing architecture for solving a sparse system of linear equations using iterative methods</td>
</tr>
<tr>
<td>80.</td>
<td>Concurrent object-oriented iterative equation solving algorithm</td>
</tr>
<tr>
<td>81.</td>
<td>A high-rise building structure</td>
</tr>
</tbody>
</table>
CHAPTER I
INTRODUCTION

1.1 General

In the mid-1960's, a new programming technology called object-oriented programming (OOP) was introduced with the development of the SIMULA-67 programming language. However, it was only after Xerox PARC (Palo Alto Research Center) introduced Smalltalk-80 in the early 1980's that OOP's popularity really took off. The reason for this popularity is due to OOP's unique characteristics in comparison with other programming paradigms; namely, data abstraction, inheritance, modularity, information hiding, and encapsulation. These characteristics satisfy many of the basic requirements for developing large and complex engineering software systems.

1.2 Related Work

The application of OOP in civil engineering is quite recent. The earliest papers on this topic are authored by Fenves [1988], Miller [1988], Powell and An-Nashif [1988] and Powell and Bhatia [1988]. Additional research has been reported since then. Powell and his students attempted to create an integrated engineering system covering architectural, structural and construction engineering [Powell and Bhatia, 1988; Powell and An-Nashif, 1988; Powell et al., 1989; Powell and Abdalla, 1989; Sause and Powell, 1990, 1991; An-Nashif and Powell, 1990, 1991; and Sause et al., 1992]. Forde et al. present an intelligent finite element analysis software system called Object NAP [Forde,
describe a finite element system in data abstraction approach by using Coordinate
Systems, Structures, Elements, Nodes, and Enforcing Constraints in the object-oriented
programming paradigm. Adeli and Hung [1990] present an object-oriented model to
process the knowledge of earthquake damage and structural behavior. Ahmed et al.
[1991b and 1992] study the coordination and communication problems in an integrated
engineering system called DICE (Distributed and Integrated Computer-Aided
Engineering Environment) through data sharing in collaborative engineering. A survey
of applications of OOP in civil engineering is summarized in Table 1. The research work
of this dissertation is check marked in the correspondent fields in Table 1.

1.3 Objectives

The research for this dissertation on the application of object-oriented
programming paradigm in computer-aided civil engineering began in 1989. With
numerical and finite element analysis particularly in mind, this research focuses on the
following objectives:

* Developing a series of graphical representation tools for conducting object-
oriented analysis, design and programming.

* Presenting a blackboard architecture for performing numerical analysis in
computer-aided engineering.

* Developing an object-oriented finite element analysis class library using database
management techniques to handle large quantities of data encountered in
computer-aided engineering.

* Studying the version problem in the software application phase rather than in the
software development phase.
Table 1. A Survey of OOP Applications in Civil Engineering

<table>
<thead>
<tr>
<th>CE</th>
<th>Methodology</th>
<th>KBES</th>
<th>Data, DBMS</th>
<th>Concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>[Rigopoulos &amp; Oppenheim, 92]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Construction</td>
<td>[Garrett &amp; Hakim, 92]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Design Standards</td>
<td>[Baugh &amp; Rehak, 89] [Forde et al., 90] [Baugh &amp; Rehak, 92] [Huang, 92]</td>
<td>[Baugh &amp; Rehak, 89] [Forde, 89]</td>
<td>[Abdalla &amp; Yoon, 92] [Huang, 92]</td>
<td>✓</td>
</tr>
<tr>
<td>FEA</td>
<td>[Fenves, 90] [Reed &amp; Chen, 91] [Ross et al., 92] [Sause et al., 92]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>General Engineering</td>
<td></td>
<td></td>
<td>[Ahmed et al., 92]</td>
<td></td>
</tr>
<tr>
<td>Geotechnical Engineering</td>
<td></td>
<td></td>
<td>[Oloufa et al., 92]</td>
<td></td>
</tr>
<tr>
<td>Integrated System</td>
<td>[Nakai et al., 91-2]</td>
<td></td>
<td>[Law et al., 90] [Ahmed et al., 91a-b] [Abdalla &amp; Yoon, 92] [Abdalla et al., 92] [Howard et al., 92]</td>
<td></td>
</tr>
<tr>
<td>Structural Engineering</td>
<td>[Fenves, 88] [An-Nashif &amp; Powell, 91]</td>
<td>[Forde &amp; Stiemer, 89] [Watson &amp; Chan, 89]</td>
<td>[Powell &amp; Bhateja, 89] [Watson &amp; Chan, 89]</td>
<td>[Miller, 89]</td>
</tr>
<tr>
<td>Test</td>
<td>[Bullock, 92]</td>
<td></td>
<td>[Law et al., 90] [Akhras &amp; Foo, 93]</td>
<td>✓</td>
</tr>
</tbody>
</table>
Proposing a data storage structure for complex engineering data objects encountered in computer-aided engineering.

Simulating concurrent object-oriented programming on a uni-processor based workstation.

Developing an equation solving algorithm using the concurrent object-oriented programming and iterative methods.

Applying the above theories to the real world problem of composite laminate analysis.

1.4 Organization of this Dissertation

This dissertation is composed of nine chapters and four appendices. The content of each chapter and appendix is described briefly in the following:

Chapter I Introduction

This chapter discusses the objectives of this dissertation research and outlines the content of each chapter.

Chapter II Global-Local Model for Composite Laminate Analysis

This chapter describes the global-local model for finite element analysis of composite laminates.

Chapter III An Architecture of Object-oriented Programming in Computer-Aided Engineering

This chapter studies the application of object-oriented programming in numerical analysis of complex engineering problems. With object-oriented programming as a unified paradigm, a controlling algorithm for numerical analysis is proposed. This algorithm
consists of blackboard architecture, extended entity-relational data model, finite element analysis, knowledge-based systems, learning systems, and concurrent programming. This chapter lays the foundation for all OOP models developed in subsequent chapters.

Chapter IV Object-Oriented Finite Element Analysis Using an EER Model
An object-oriented finite element modeling approach is presented for the solution of complex engineering systems. For efficient processing of a myriad of data types generated in such analyses, an object-oriented enhanced entity relationship model is developed. A class library is created for performing object-oriented finite element analysis in C++. These classes model the basic concepts and tools needed for finite element analysis of engineering problems. As an example, the models and class library developed in this research are applied to interlaminar stress analysis of composite structures.

Chapter V An Object-Oriented Data Management Model
This chapter presents an object-oriented data management model for numerical analysis in computer-aided engineering. A data storage structure and an index system have been developed for complex data objects encountered in numerical analysis. As an example, the model is applied to the finite element analysis.

Chapter VI An Object-Oriented Version Management Model
This chapter addresses the problem of object version management in applying object-oriented programming to numerical analysis in computer-aided engineering. The existing version management models are reviewed and the necessity of developing a special version management model embedded in CAE applications is indicated. A new version management model that is particularly effective for computer-aided engineering is
proposed. This version management model is based on the concept of parallel version graph which incorporates the nature of design objects with alternative data in computer-aided engineering.

Chapter VII  A Concurrent Approach to Object-Oriented Finite Element Analysis

In this chapter, a concurrent object-oriented programming approach to finite element analysis is presented. Although concurrent object-oriented programming languages are still under development, various concurrent object-oriented programming techniques simulated by the UNIX operating system are discussed in this chapter.

Chapter VIII Concurrent Object-Oriented Iterative Equation Solver

This chapter discusses a new concurrent object-oriented iterative equation solver. The recognition of vector operations in iterative methods and more zeroes than non zero elements in the stiffness matrix of a large structure leads us to explore a new approach for solving a system of linear equations, that is, the object-oriented programming approach to define sparse vector operations, and to apply parallel processing techniques in performing iterative equation solving methods. The consistent, independent and relatively simple operations of sparse vector objects in this algorithm allow for increased speedup. As an example, the concurrent object-oriented iterative equation solver is then used to solve a large system of linear equations generated in structural analysis of a highrise building structure.

Chapter IX Conclusions and Recommendations

This chapter summarizes this dissertation research and provides some recommendations for future study in this area.
List of References
This section is a bibliography of all references directly and indirectly influencing this research. The author's published and in-review papers are listed as well.

Appendix A
This appendix presents a class library of vector computation in C++ object-oriented programming language.

Appendix B
This appendix presents a class library of matrix computation in C++ object-oriented programming language.

Appendix C
This appendix presents a class library of object-oriented finite element analysis in C++ object-oriented programming language.

Appendix D
This appendix presents the source code for composite laminate analysis.

Appendix E
This appendix includes the Dissertation Abstract to be published in Dissertation Abstracts International by University Microfilms, Inc.

Appendix F
This appendix presents an index of the Dissertation.
CHAPTER II
GLOBAL-LOCAL MODEL FOR COMPOSITE LAMINATE ANALYSIS

2.1 Introduction

The advanced fiber reinforced polymer composite materials offer several advantages such as light weight, high specific stiffness and strength properties and design flexibility. Development of accurate, efficient and reliable stress analysis procedures for the laminated composites has caught the attention of the research community.

One important problem in the analysis of composite laminate is the so-called free-edge problem. The interlaminar stresses generated around the free-edges and in-between laminae are recognized to be the primary sources causing delamination of composite laminates initiated at the free-edges. Many approaches have been suggested to analyze this problem since Hayashi's pioneer work [1967]. Pipes and Pagano [1970] use elasticity formulation to model free-edge stresses, and obtain solution for interlaminar stresses through finite-difference techniques. Pagano and Pipes [1973] present several observations based upon their experiments which later become the basis for further analytical and numerical studies of free-edge delamination problems. It is observed that the interlaminar normal stress remains compressive till near the vicinity of free-edge, and changes to tension with large value. This tensile normal stress causes delamination of adjacent plies near the free-edge.
Studies of Wang and Crossman [1977] on this problem can be summarized as the following: i) normal stress arises at the free edge; ii) normal stress at the free edge is not infinite; iii) signs and absolute values of laminates in (90/0)s and (0/90)s stack sequences are different.

Based on Reissner's variational principle [1950] and the assumptions that the in-plane stresses are linear functions of depth z within each layer, Pagano [1978] proposes a new theory to determine interlaminar stress field in a composite laminate. The results from this theory are in close agreement with those of Wang and Grossman [1977].

Wang and Choi [1982a, 1982b] present a mathematical model and its solutions to the free-edge problem based on Lekhnitskii's stress potentials and the anisotropic elasticity theory. Their conclusions include i) geometric and material discontinuities cause the singularity of interlaminar stresses; ii) the singularity depends on material properties and fiber orientations of adjacent layers; and iii) the singularity is of lower order in composite laminates than that observed in isotropic solid materials.

A large body of literature exists on the free-edge problem. It can be observed that as the number of plies in a laminate increases, many analytical/numerical models can become either too complicated to handle, or much computer time consuming for many real-life practical applications. Moreover, many times the composite material users may be interested in obtaining interlaminar stress and/ or strain results only at a few interlaminar planes but not at all the planes in a laminate. To practically manage some of the above difficulties and requirements, Pagano and Soni [1983] propose a model called global-local model in which the entire laminate thickness can be divided into two convenient regions: one is called global which is assumed to be a single unit, while the other region, the local one, regards each interlaminar plane as distinct within its boundaries. This concept is selective as far as looking at only the interlaminar regions of user's interest, and appears to give reasonably good results.
2.2 The Global-Local Model

The basic idea of the global-local model is division of a laminate into two regions, a global region and a local region. The concept is shown in Figure 1. The local region has $N$ layers, and the global region has $(N_T - N)$ layers, in which $N_T$ is the total ply number. The thickness of layer $i$ in the local region is denoted by $h_i$. The total thickness of the global region is denoted by $H$.

In the local region, the model developed by [Pagano. 1978] is applied. In the global region, the model is based on a higher order laminate theory [Whitney and Sun. 1973]. It is represented by a potential energy integral to be consistent with the local region formulation. The consistency of two regions is guaranteed by enforcing appropriate continuity conditions of interlaminar stresses and displacements.

![Figure 1. The global-local model](image)
The total potential energy of the composite laminate can be expressed as:

\[ \pi = \pi_{\text{global}} + \pi_{\text{local}} \]  

where \( \pi_{\text{global}} \) is the potential energy of the global region, and \( \pi_{\text{local}} \) is the Reissner's mixed variational functional for \( N \) layers of the local region. They can be further expressed in Eqs. (2)-(3) as follows:

\[ \pi_{\text{global}} = \int_{V_{\text{global}}} W_p(u_i, e_{ij}) dV \]  

(2)

\[ \pi_{\text{local}} = \sum_{k=1}^{N} \int_{V_{\text{local}}} \left[ \frac{1}{2} \sigma_{ij}(u_{ij} + u_{ij,t}) - W_c(\sigma_{ij}, e_{ij}) \right] dV_k - \int_{S} \tau_i u_i dS \]  

(3)

In the above equations, \( W_p \) and \( W_c \) stand for the potential energy intensity (per unit volume) for the global region and the complementary energy intensity (per unit volume) for the local region, respectively. The terms \( V_{\text{global}} \) and \( V_{\text{local}} \) refer to the volume domains of the global and local regions, respectively.

2.3 Discretization of Global-Local Model

In this section, we will describe a finite element approach to the global-local model [Chaturvedi et al., 1992]. We define the constitutive laws as [Pagano, 1978] as follows:

\[ \sigma_i = C_{ij}(\varepsilon^o_{ij} + z\kappa^o_j - e_{ij}) \]  

(4)

\[ \sigma_i = C_{ij}\left(\varepsilon^o_{ij} + z\kappa^o_j + z^2 \frac{1}{2} \beta^o_j \right) \]  

(5)

\( i, j = 1, 2, 3, 6 \)  

\( i, j = 4, 5 \)
where $\sigma_i$ is the stress tensor; $C_i$ is the stiffness component; $\varepsilon^0_j$ is the mid-surface strains; $\varepsilon_j$ is the pre-strains; $z$ is the distance from the mid-surface; and $\kappa_j$ and $\beta_j$ are curvature parameters characterizing strains at $z$. Stress components $\sigma_1$, $\sigma_2$, $\sigma_3$, $\sigma_4$, $\sigma_5$, and $\sigma_6$ stand for $\sigma_x$, $\sigma_y$, $\sigma_z$, $\tau_{yz}$, $\tau_{xz}$, and $\tau_{xy}$, respectively.

$$\begin{pmatrix} \sigma_1 & \sigma_2 & \sigma_3 & \sigma_4 & \sigma_5 & \sigma_6 \end{pmatrix} = \begin{pmatrix} \sigma_x & \sigma_y & \sigma_z & \tau_{yz} & \tau_{xz} & \tau_{xy} \end{pmatrix}$$ (6)

The strains $\varepsilon^0_j (j=1, 2, 3, 6)$ represent the engineering strain components corresponding to $\sigma_x$, $\sigma_y$, $\sigma_z$, $\tau_{xy}$, and $\sigma_6$ ($\tau_{xz}$), and $\varepsilon^0_j (j=4, 5)$ the interlaminar strains corresponding to $\sigma_4$ ($\tau_{yz}$) and $\sigma_5$ ($\tau_{xz}$). These stress and strain components are shown in Figure 2.
Figure 2. Stress and strain definition
2.3.1 Discretization of Global Region

The potential energy density in global region due to applied loading can be expressed as the following:

\[ W_p = \frac{1}{2} \sigma \varepsilon = \frac{1}{2} (\sigma_1 \varepsilon_1 + \sigma_2 \varepsilon_2 + \sigma_3 \varepsilon_3 + \sigma_4 \varepsilon_4 + \sigma_5 \varepsilon_5 + \sigma_6 \varepsilon_6) \]  

(7)

By substituting Eqs. (4) and (6) into Eq. (2), and re-arranging the terms with respect to \( C_{ij} \), we obtain the following:

\[
\pi_{\text{global}} = \frac{1}{2} \int_{V_{\text{global}}} \left( \pi_{g1} + z^2 \pi_{g2} + z^3 \pi_{g3} + z^4 \pi_{g4} \right) \, dV
\]

\[
= \frac{1}{2} \int_{A_{\text{global}}} \left( h\pi_{g1} + \frac{h^3}{12} \pi_{g3} + \frac{h^5}{80} \pi_{g5} \right) \, dA
\]

(8)

where

\[ \pi_{g1} = \varepsilon^\sigma \overline{C} \varepsilon^\sigma + \varepsilon^{\sigma^\sigma} \overline{C}_2 \varepsilon^{\sigma^\sigma} + 2 \varepsilon^{\sigma^\sigma} \overline{C}_2 \varepsilon^{\sigma^\sigma} \]  

(9)

\[ \pi_{g2} = 2 \kappa^\tau \overline{C} \kappa^\tau + 2 \kappa^{\sigma^\sigma} \overline{C}_2 \kappa^{\sigma^\sigma} + 2 \kappa^{\sigma^\sigma} \overline{C}_3 \kappa \]  

(10)

\[ \pi_{g3} = \kappa^\tau \overline{C}_2 \kappa + \kappa^{\sigma^\sigma} \overline{C}_2 \kappa + \varepsilon^{\sigma^\tau} \overline{C}_2 \beta + \beta^\tau \overline{C}_2 \kappa + \beta^{\sigma^\sigma} \overline{C}_3 \kappa \]  

(11)

\[ \pi_{g4} = \beta^\tau \overline{C}_2 \kappa \]  

(12)

\[ \pi_{g5} = \frac{1}{4} \beta^\tau \overline{C}_2 \beta \]  

(13)

The terms with \( \pi_{g2} \) and \( \pi_{g4} \) will become zero after integration, because of odd order of \( z \).

Strains and curvatures in Eq. (7) are grouped as \( \varepsilon^\sigma, \varepsilon^{\sigma^\sigma}, \kappa, \kappa^{\sigma^\sigma} \) and \( \beta \). They are defined in [Soni et al., 1989]. The effective global region material stiffness matrix is given by [Tsai, 1987]. We separate the stiffness matrix \( C \) into three sub-matrices \( \overline{C}_1, \overline{C}_2 \) and \( \overline{C}_3 \):
Because the material is monoclinic, the entities with $\mathbf{C}_3$ turn to be zero.

In order to discretize parameters in the analysis, the generalized displacements $\bar{u}$, $\bar{v}$, $\bar{w}$, $u^*$, $v^*$, $w^*$, and $\hat{w}$ are introduced in terms of $u$, $v$ and $w$, which are the displacements along $x$, $y$ and $z$ directions [Soni et al., 1989]. For the convenience of mathematical manipulation, we place strains and curvatures together with interlaminar stresses $p_1 (\sigma_z)$, $s_1 (\tau_{yz})$, and $t_1 (\tau_{xz})$ in the following equation. Subscripts 1 and 2 stand for bottom and top of the layer in a laminate.

$$\mathbf{u}^T = \begin{bmatrix} \bar{u} & \bar{v} & \bar{u}^* & \bar{v}^* & \bar{w} & \bar{w}^* & p_1 & p_2 & s_1 & s_2 & t_1 & t_2 \end{bmatrix}$$ (17)

Let us assume the displacements in composite laminates $u$, $v$, and $w$ to be of the following form [Whitney and Sun, 1973]:

$$u = u^0(x,y) + z\varphi_x(x,y)$$ (18)

$$v = v^0(x,y) + z\varphi_y(x,y)$$ (19)

$$w = w^0(x,y) + z\varphi_z(x,y) + \frac{z^2}{2}\Phi(x,y)$$ (20)
The expressions for $u$, $v$, $w$, and $w_o$ in terms of generalized displacement can be obtained as follows:

$$u = \frac{\bar{u}}{2} + z \frac{3u^*}{H} \tag{21}$$

$$v = \frac{\bar{v}}{2} + z \frac{3v^*}{H} \tag{22}$$

$$w = \frac{9}{8} \frac{\bar{w}}{2} - \frac{15}{8} \frac{\dot{w}}{2} + z \frac{3}{H} \frac{w^*}{2} + \frac{z^2}{2} \frac{45}{H^2} (\dot{w} - \ddot{w}) \tag{23}$$

$$w_o = \frac{9}{8} \frac{\bar{w}}{2} - \frac{15}{8} \frac{\dot{w}}{2} \tag{24}$$

In the final analysis, it is necessary to maintain the continuity of displacements $u$, $v$ and $w$ between neighboring layers of the laminate. For this purpose, we choose displacements at the top and bottom of each layer as nodal parameters denoted by subscripts $t$ and $b$, respectively.

$$u_{tb}^T = \left\{ u_t, v_t, w_t, p_{2t}, s_{2t}, t_2, w_o, u_b, v_b, w_b, p_{1b}, s_1, t_1 \right\} \tag{25}$$

The seven unknown nodal parameters are shown in Figure 3. They can be obtained by substituting $z = \pm \frac{H}{2}$ into Eqs. (21)-(24). The matrix expression of $u_{tb}$ is Eq. (26), where $A$ is an operator matrix [Chaturvedi et al., 1992].

$$u = A \quad u_{tb} \tag{26}$$

By interpolating nodal variables, one can obtain the expression for $u_{tb}$ as follows:

$$u_{tb} = N_c \quad u_c^X \tag{27}$$

where $N_c$ is a shape function matrix, and

$$u_c^X = \left\{ u_t, v_t, w_t, p_{2t}, s_{2t}, t_2, w_o, u_b, v_b, w_b, p_{1b}, s_1, t_1 \right\} \tag{28}$$
Each sub-vector of $N^e$ has the same number of elements as interpolation nodes. For instance, in the case of 8-node interpolation, sub-vector has 8 elements, such that $u^e$ has 104 entities in total.

Figure 3. Illustration of the interlaminar nodal displacements
Combining Eqs. (20) and (21), the unknown nodal variable vector \( u \) becomes:

\[
u = A \ u_n = A \ N_c \ u_c^N = A_c \ u_c^N
\]  

(29)

where \( A_c \) is a matrix whose each entity is a vector of size being equal to the number of interpolation nodes as follows:

\[
A_c = \begin{bmatrix}
N & 0 & 0 & 0 & 0 & 0 & 0 & N & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & N & 0 & 0 & 0 & 0 & 0 & 0 & N & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{3}N & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{3}N & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{3}N & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{3}N & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{1}{3}N & 0 & 0 & 0 & 0 & \frac{1}{3}N & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{1}{3}N & 0 & 0 & 0 & 0 & \frac{1}{3}N & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{1}{5}N & 0 & 0 & 0 & \frac{4}{15}N & 0 & 0 & \frac{1}{5}N & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & N & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & N & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & N & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & N & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & N & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]  

(30)

Now we define strains and curvatures in terms of \( u \) as:

\[
\varepsilon^o = L_{12}u = L_{12}A_c u_c^N = L_{12N}u_c^N
\]  

(31)

\[
\bar{\varepsilon}^o = \bar{L}_{12}u = \bar{L}_{12}A_c u_c^N = \bar{L}_{12N}u_c^N
\]  

(32)

\[
\kappa = L_{55}u = L_{55}A_c u_c^N = L_{55N}u_c^N
\]  

(33)
\[ \bar{K} = \bar{L}_{36} \mu = \bar{L}_{34} A_{\xi} u_{\xi} = \bar{L}_{345} u_{\xi} \]  

(34)

\[ \beta = L_{56} \mu = L_{56} A_{\xi} u_{\xi} = L_{565} u_{\xi} \]  

(35)

where

\[
\bar{L}_{12} = \begin{bmatrix}
\frac{1}{2} \frac{\partial}{\partial x} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{2} \frac{\partial}{\partial y} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 3 \frac{h}{\partial} & 0 & 0 & 0 \\
1 \frac{\partial}{\partial y} & 1 \frac{\partial}{\partial x} & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]  

(36)

\[
\bar{L}_{32} = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{9}{8} \frac{\partial}{\partial y} & \frac{-15}{8} \frac{\partial}{\partial y} & 0 & 0 \\
0 & 0 & 3 \frac{h}{\partial} & 0 & \frac{9}{8} \frac{\partial}{\partial x} & \frac{-15}{8} \frac{\partial}{\partial x} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{\partial}{\partial x} & 0
\end{bmatrix}
\]  

(37)

\[
\bar{L}_{34} = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{\partial}{\partial y} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]  

(38)

\[
L_{32} = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{3}{h} \frac{\partial}{\partial x} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{-15}{h^2} & 0 \\
0 & 0 & \frac{3}{h} \frac{\partial}{\partial y} & \frac{3}{h^2} \frac{\partial}{\partial x} & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]  

(39)

\[
L_{56} = \begin{bmatrix}
0 & 0 & 0 & 0 & 0 & \frac{-15}{h^2} \frac{\partial}{\partial y} & 0 & \frac{45}{h^2} \frac{\partial}{\partial y} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{-15}{h^2} \frac{\partial}{\partial x} & 0 & \frac{45}{h^2} \frac{\partial}{\partial x} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{\partial}{\partial x} & 0
\end{bmatrix}
\]  

(40)
Finally, Eq. (7) becomes Eq. (41). The items inside the bracket contribute to the stiffness matrix of the global region.

\[
\pi_g = \frac{1}{2} u_t^{\text{global}} \left[ \begin{array}{c}
\int_{A_{\text{global}}} \left( h L_{12N}^T \mathbf{C}_1 L_{12N} + h L_{12N}^T \mathbf{C}_2 L_{12N} \\
+ 2 h L_{12N}^T \mathbf{C}_3 L_{12N} + \frac{h^3}{12} L_{56N}^T \mathbf{C}_1 L_{56N} \\
+ \frac{h^3}{12} L_{56N}^T \mathbf{C}_3 L_{56N} + \frac{h^3}{6} L_{34N}^T \mathbf{C}_3 L_{34N} \\
+ \frac{h^5}{320} L_{56N}^T \mathbf{C}_2 L_{56N}
\end{array} \right] dA \right] u_t^{\text{global}}
\]

In Eq. (41), one entity \( \frac{h^3}{12} L_{12N}^T \mathbf{C}_2 L_{56N} \) is not symmetrical. As all entities with \( \mathbf{C}_3 \) can be ignored by virtue of assumed material symmetry, we can conclude that the global region stiffness matrix is unsymmetrical. This causes the total stiffness matrix of global-local model to be unsymmetrical.

### 2.3.2 Local Region

For simplicity, we can write Eq. (3) as Eq. (42).

\[
\pi_{\text{local}} = \int_{V_{\text{local}}} \left[ \sigma^T (\varepsilon + \mathbf{e}) - W_c \right] dV
\]

By further replacing terms of Eq. (42) with stress-strain relationship, we can represent the Reissner's mixed variational functional as:
\[
\pi_{\text{local}} = \int_{V_{\text{local}}} \left[ \sigma^T (\varepsilon + e) - W \right] \, dV
\]
\[
= \int_{V_{\text{local}}} \left[ \sigma^T (S\sigma - e + e) - \frac{1}{2} \sigma^T S\sigma + \sigma^T e \right] \, dV
\]
\[
= \int_{V_{\text{local}}} \left[ \frac{1}{2} \sigma^T S\sigma + \sigma^T e \right] \, dV
\]

(43)

Similar to the case of strain components for the global region, the stress components for the local region are divided into two vectors for mathematical convenience, and they are

\[
\sigma_{1}^{T} = \{\sigma_{1}, \sigma_{2}, \sigma_{3}, \sigma_{6}\}
\]
\[
(44)
\]
\[
\sigma_{2}^{T} = \{\sigma_{4}, \sigma_{5}\}
\]
\[
(45)
\]
such that the Eq. (43) can be re-written as:

\[
\pi_{\text{local}} = \pi_{\text{local}1} + \pi_{\text{local}2} + \pi_{\text{local}3}
\]
\[
(46)
\]

where

\[
\pi_{\text{local}1} = \frac{1}{2} \int_{V} \sigma_{1}^{T} \bar{S}_{1} \sigma_{1} \, dV
\]
\[
(47)
\]
\[
\pi_{\text{local}2} = \frac{1}{2} \int_{V} \sigma_{2}^{T} \bar{S}_{2} \sigma_{2} \, dV
\]
\[
(48)
\]
\[
\pi_{\text{local}3} = \frac{1}{2} \int_{V} \sigma_{2}^{T} \bar{S}_{3} \sigma_{3} \, dV
\]
\[
(49)
\]

\[
\bar{S}_{1} = \begin{bmatrix}
S_{11} & S_{12} & S_{13} & S_{16} \\
S_{12} & S_{22} & S_{23} & S_{26} \\
S_{13} & S_{23} & S_{33} & S_{36} \\
S_{16} & S_{26} & S_{36} & S_{66}
\end{bmatrix}
\]
\[
(50)
\]

\[
\bar{S}_{2} = \begin{bmatrix}
S_{44} & S_{45} \\
S_{45} & S_{55}
\end{bmatrix}
\]
\[
(51)
\]
\[
\mathbf{S}_{21} = \mathbf{S}_{12}^T = \begin{bmatrix} S_{11} & S_{12} & S_{13} & S_{14} \\ S_{15} & S_{22} & S_{23} & S_{24} \\ S_{31} & S_{32} & S_{33} & S_{34} \\ S_{41} & S_{42} & S_{43} & S_{44} \end{bmatrix}
\]

(52)

In Eq. (46), \( \pi_{\text{local}} \) will be zero if the material is monoclinic. In the case of orthotropic composite layer, the coupling part \( \pi_{\text{local}} \) of Eq. (46) can be neglected, such that

\[
\pi_{\text{local}} = 1 \int \sigma_{ij}^T \mathbf{S}_{ij} \sigma_{kl}^l \, dV + 1 \int \sigma_{ij}^T \mathbf{S}_{ij} \sigma_{kl}^2 \, dV
\]

(53)

We assume that the stresses \( s_1, s_2 \) and \( s_6 \) are linear functions of coordinate \( z \) as follows:

\[
\sigma_1 = \sigma_x = \frac{N_x}{h} + \frac{12M_1 z}{h^3}
\]

(54)

\[
\sigma_2 = \sigma_y = \frac{N_y}{h} + \frac{12M_2 z}{h^3}
\]

(55)

\[
\sigma_6 = \tau_{xy} = \frac{N_{xy}}{h} + \frac{12M_{xy} z}{h^3}
\]

(56)

For stresses of \( s_3, s_4 \) and \( s_5 \), we can substitute Eq. (43) into the differential equations of equilibrium along with the values of the interlaminar stresses at \( z = \pm h/2 \) and obtain:

\[
\sigma_3 = \sigma_z = \frac{p_1 + p_2}{4} \left( \frac{12z^2}{h^2} - 1 \right) + \frac{p_2 - p_1}{4} \left( \frac{40z^3}{h^3} - 6z \right)
+ \frac{3N_x}{2h} \left( \frac{1 - 4z^2}{h^2} \right) + \frac{15M_x}{2h} \left( \frac{2z - 8z^3}{h^3} \right)
\]

(57)

\[
\sigma_4 = \tau_{yz} = \frac{z}{h} \left( \frac{s_2 - s_1}{4} \right) \left( \frac{12z^2}{h^2} - 1 \right) + \frac{3V_y}{2h} \left( 1 - 4z^2 \right)
\]

(58)

\[
\sigma_5 = \tau_{xz} = \frac{z}{h} \left( \frac{t_2 - t_1}{4} \right) \left( \frac{12z^2}{h^2} - 1 \right) + \frac{3V_x}{2h} \left( 1 - 4z^2 \right)
\]

(59)
These vectors in Eqs. (54) through (59) can be written in terms of the force and moment components and the interlaminar stress components. They are

\[
\sigma^1 = M^* \quad N_R
\]

\[
\sigma^2 = M^* \quad V
\]

where

\[
N_R = \begin{bmatrix} N_x & N_y & N_z & N_{xy} & N_{xz} & N_{yz} & M_x & M_y & M_z & M_{xy} & M_{xz} & M_{yz} & p_1 & p_2 \end{bmatrix}
\]

\[
V = \begin{bmatrix} V_x & V_y & s_1 & s_2 & t_1 & t_2 \end{bmatrix}
\]

\[
M^* = \begin{bmatrix} h & 0 & 0 & 0 & 12z \frac{h}{h} & 0 & 0 & 0 & 0 \end{bmatrix}
\]

\[
M^* = \begin{bmatrix} 0 & h & 0 & 0 & 0 & 12z \frac{h}{h} & 0 & 0 & 0 \end{bmatrix}
\]

\[
M^* = \begin{bmatrix} 0 & 0 & F_1 & 0 & 0 & 0 & F_2 & 0 & F_3 & F_4 \end{bmatrix}
\]

\[
M^* = \begin{bmatrix} 0 & 0 & 0 & h & 0 & 0 & 0 & 12z \frac{h}{h} & 0 & 0 \end{bmatrix}
\]

\[
M^* = \begin{bmatrix} 0 & F_1 & F_2 & F_3 & F_4 & 0 & 0 \end{bmatrix}
\]

\[
M^* = \begin{bmatrix} F_1 & 0 & 0 & 0 & F_5 & F_6 \end{bmatrix}
\]

where

\[
F_1 = \frac{3h}{2} - 6z^2 \frac{h}{h}
\]

\[
F_2 = \frac{30z}{h} - 120z^3 \frac{h^3}{h}
\]

\[
F_3 = \frac{-1}{4} + \frac{3z}{2h} + \frac{3z^2}{h^2} - 10z^3 \frac{h^3}{h}
\]

\[
F_4 = \frac{-1}{4} - \frac{3z}{2h} + \frac{3z^2}{h^2} + 10z^3 \frac{h^3}{h}
\]
\[
F_5 = -\frac{1}{4} - \frac{z}{h} + \frac{3z^2}{h^2} 
\]  \quad (70)

\[
F_6 = -\frac{1}{4} + \frac{z}{h} + \frac{3z^2}{h^2} 
\]  \quad (71)

In the above expressions, variables \( V_x \) and \( V_y \) are the shear resultants, \( N_z \) and \( M_z \) are out-of-plane force and moment resultants, and \( h \) is the thickness of the layer under consideration.

Now we have expression of Eq. (56) in terms of strains and forces:

\[
\pi_{\text{local}} = \frac{1}{2} \int \left( \begin{array}{cccc}
N_R^T & M_1^T & \overline{S}_{11} & M_1^* \\
M_1 & N_R + V^T M_2^T & \overline{S}_{22} & M_2^* \\
\overline{S}_{11}^* & \overline{S}_{22}^* & \overline{V} & \overline{V} \\
\end{array} \right) dV 
\]  \quad (72)

Considering the constitutive relationships between two adjacent layers, it is reasonable to express \( N_R \) and \( V \) in terms of displacements and interlaminar stresses at the nodal point, which are described in Eq. (17).

\[
N_R = R_1 u 
\]  \quad (73)

\[
V = R_2 u 
\]  \quad (74)

As \( \overline{S}_{11} \) and \( \overline{S}_{22} \) are independent of \( x, y \), and \( z \), and \( M_1^* \) and \( M_2^* \) are functions of coordinate \( z \), one can define the following:

\[
\overline{S}_{11}^* = \int \overline{M}_1^T \overline{S}_{11} \overline{M}_1^* dz 
\]  \quad (75)

\[
\overline{S}_{22}^* = \int \overline{M}_2^T \overline{S}_{22} \overline{M}_2^* dz 
\]  \quad (76)

Now, Eq. (53) can be rewritten as Eq. (77):

\[
\pi_{\text{local}} = \frac{1}{2} \int_{\Omega_{\text{local}}} \left[ \begin{array}{cccc}
R_1 & u^T & \overline{S}_{11}^* & R_1 \\
\overline{S}_{11}^* & \overline{S}_{22}^* & \overline{R}_2 & \overline{R}_2 \\
\end{array} \right] dA 
\]  \quad (77)
To obtain matrices \( R_1 \) and \( R_2 \), we use the elastic constitutive relations and the
differential equations of equilibrium described in [Pagano, 1978]. The detailed
derivations and expressions for \( R_1 \) and \( R_2 \) are not presented for the sake of space.

2.4 Assembly of Global And Local Region Stiffness Matrix

There exist thirteen unknown variables \( \mathbf{u}_{\text{ub}} \) [Eq. (25)] which determine that the
stiffness matrix is a 13x13 matrix (for finite element discretization, it is more accurate to
say that the matrix is 13*NNODE by 13*NNODE, where NNODE stands for
interpolation node number). By considering appropriate continuity between two adjacent
layers, the entities can be assembled as shown in Figure 4.

The algorithm of assembling stiffness matrix is described as follows:

a) calculate stiffness matrix of 1\(^{st} \) layer and store in \( [T] \);
b) write first 7-rows of \( [T] \) on the data file;
c) restore/transfer rest of \( [T] \) to \( [T_1] \) (6x7) and \( [T_2] \) (6x6). \( [T_2] \) is the interrelated
   part to layer 2;
d) calculate stiffness matrix of 2\(^{nd} \) layer and store in \( [T] \);
e) write 6 rows of \( [T_1] \) on to the data file, together with \( [T] \). The first 6x6
   submatrix of \( [T] \) is added by \( [T_2] \).
f) write 7\(^{th} \) row of \( [T] \) to the data file (7\(^{th} \) row refers to \( w_0 \) only);
g) if it is not the last layer, go to step c) with increasing of layer number;
h) else, write rest 7-rows of \( [T] \) on the data file.

The above assembly procedure is based on the assumption that global region is at
the top of a laminate. If not, the assembly algorithm needs further modification. Since
global region can be treated as one special layer, the assembly process is accomplished during the computation of local region stiffness matrix.

Figure 4. Assembly of element stiffness matrix
CHAPTER III

AN ARCHITECTURE OF OBJECT-ORIENTED PROGRAMMING IN COMPUTER-AIDED ENGINEERING

3.1 Introduction

Computers are playing a key role in assisting numerical analysis for scientists and engineers in solving complex engineering problems, such as orbit calculation for space shuttles, earthquake resistance design of high-rise buildings, computer simulation of wind tunnel test and so on. Although both hardware and software technologies have improved greatly in recent years, the way engineers use computers is still the same as a few decades ago: deriving a series of equations, preparing flowcharts and programming one line after another according to these derived equations. With the growing complexity of problems, software developers realize that the conventional waterfall algorithm-driven structured programming becomes less effective.

The object-oriented programming (OOP) technique developed in software engineering community is a logic extension of the structured programming [Tockey et al., 1990]. The applications of OOP in computer-aided engineering (CAE) is also under active research in civil engineering [Forde and Stiemer, 1989; Fenves, 1990; Miller, 1991; Nakai, 1992; Stiemer and Felber, 1993; only name a few]. However, most of researches are focusing on design process rather than numerical analysis. In fact, the numerical analysis in modern design procedure becomes more complicated than ever. For example, a good finite element analysis (FEA) software system might contain tens of thousands lines of code and hours of CPU time to obtain final results. At the same time.
team work becomes a common practice in developing complex software systems. The cooperation among team members are crucial to the success of software development. The object-oriented programming provides techniques for solving a complex problem in a team effort extensively.

The blackboard architecture (BB) developed in artificial intelligence (AI) community shows its potential in organizing general software systems other than artificial intelligence. It appears that the object-oriented programming coincides with the blackboard architecture in the perspective of software system structure. This chapter presents an architecture for numerical analysis in computer-aided engineering based on the blackboard architecture using object-oriented programming paradigm. The blackboard is managed using the technology of object-oriented database management system (OODBMS). The modularity achieved by applying OOP greatly benefits the concurrent execution of the software system. As an application, we present a complex stress analysis problem in composite laminates.

3.2 Blackboard Architecture

The blackboard architecture was first introduced by Newell [1962] and implemented in Hearsay-II project for speech recognition. The latest version, Hearsay-III [Erman et al., 1988], takes off the time dimension of Hearsay-II and makes the blackboard architecture to be a domain-independent problem-solving structure. Since then, its concept has been applied to various knowledge-based systems in engineering [Adeli, 1990] [Aravind et al., 1991].

The blackboard architecture (Figure 5) is composed of three major components: knowledge source (KS), blackboard (BB), and controller (also called world model). The knowledge source can be a knowledge-based system, a neural network learning system, a
Figure 5. Framework of blackboard architecture
solution algorithm, a group of design criteria and a combination of them. In the case of a knowledge-based system, it is further divided into two sub-components: rules (domain knowledge) and an inference engine to fire these rules. The inference engine can be made of another knowledge source. Thus, the entire knowledge source is composed of a knowledge source hierarchy to represent heuristics in application problems.

The knowledge source is activated sequentially without explicit order. There is no interruption until one KS is completely activated. As the trial and error goes on, the hypothesis on the blackboard is getting converged. This is illustrated by evolution from ellipses to a circle in Figure 5. During the process, there is no guarantee each KS is moving forward and getting closer to the final results. The subsequent KS works on the basis of the former one. If one KS finds out the mistake made by the former KS, it corrects the mistake and builds up a new basis for next trial. The whole process of activating knowledge sources is a problem-solving cycle, and similar to the jigsaw puzzle game [Englemore and Morgan, 1988]. Each KS participates in the process. The result comes from the cooperation of efforts from all knowledge sources.

The blackboard is a shared database for posting and modifying hypotheses among knowledge sources. Independent knowledge sources use this platform to exchange information and interact with each other. When one knowledge source is activated by the controller, it reads the information from the blackboard, compares the information with its knowledge and selects an action: either modifies the existing hypothesis on the blackboard or creates a new hypothesis. As the cycle of activating KS continues, it is getting closer to the solution of the problem.

Blackboard architecture is the natural exploitation of concurrency. It encourages parallelism in knowledge sources, information transferring, and data partitioning [Nii et al., 1988]. In reality, however, many implementation cases of blackboard architecture are still sequential execution of knowledge sources [Lamer, 1990].
The blackboard architecture has many advantages in software development. Aravind et al. [1991] and Lander [1992] list them as follows:

a) organizing knowledge into independent knowledge modules;
b) representing knowledge in each module differently;
c) using different inference engine in each module;
d) being a flexible representation structure to various applications;
e) using levels of abstraction;
f) allowing co-existence of inconsistent solutions; and
g) developing software system incrementally by using modules.

In conclusion, the blackboard architecture has the flexibility for general purpose application and the modularity for developing software systems incrementally and activating modules concurrently.

3.3 Principles of Blackboard Architecture for Numerical Analysis

The blackboard architecture is a domain-independent concept for general software development. We propose a new architecture for numerical analysis based on the blackboard architecture, called Blackboard Architecture for Numerical Analysis (BANA). This new architecture broadens the blackboard architecture and adds more detail functions to perform the complex numerical computation in solving engineering problems. The basic elements of the proposed architecture for numerical analysis are as follows: module, controller, knowledge source, global level, local level, blackboard, controlling data slot, and numerical data slot. They are shown in Figure 6.
Figure 6. General blackboard architecture for numerical analysis
This architecture is primarily composed of a set of modules and a group of blackboards. Modules are placed at two levels. One is the global level, and another one is the local level. The global level has a supreme controller and its assistant knowledge sources. They dominate all other modules at the local level. Modules at the local level can be further divided according to their dependency. The module at the local level can control another module at its sub-local level. As long as it is necessary, there is no limitation on the expansion of sub-levels. This reflects the fact that in the real world, many computation procedures are related with each other in a hierarchy of dependency, and complex problems are usually decomposed into less complex sub-problems.

Another advantage of arranging modules into two levels is to conduct computation concurrently. Each module is a task and executed on one computer processor as a process. The module at the global level takes in charge of the whole system, and assigns those independent modules at the local level to processors to be executed concurrently on the massively parallel computer. For example, the learning system implemented in neural networks can benefit from this structure, because many neuron-like processing elements compute simultaneously. For a typical numerical analysis, the finite element analysis, it is obvious to compute element stiffness matrices in parallel to speed up computation process.

Within each module, there are a controller and one or more knowledge sources. The function of the controller is to activate knowledge sources in a certain sequence and make evaluation about results (hypotheses) shown on the blackboard. The controller is the headquarters of the entire module. After the task assigned to the module is accomplished, the controller terminates the activity of the module and sends back a signal to the controller in the upper level. In case of difficult situations, the controller itself is monitored by engineers through user interface of the computer. Valuable advice is input to the controller to help system’s performance whenever it is necessary.
The knowledge source is an open concept and not limited to the knowledge defined in artificial intelligence technology. We consider mathematical equations, industrial standards, and test data as knowledge as well in this architecture. Fundamentally, they are all wisdom, and obtained from either experiment or experience of scientists and engineers in their practice.

There might be many different ways to obtain the same numerical results in engineering practices. Some of them perform numerical computation faster than others, or provide more reasonable solutions to the problem. To handle different situations wisely, the proposed architecture includes the knowledge-based systems and learning systems in addition to mathematical equations, industrial standards, and test data in the knowledge sources. This makes the proposed architecture smarter. For example, in the finite element analysis, hundreds or thousands of elements are created on the problem domain. Each element has one stiffness matrix to be computed. If the knowledge source is able to detect the equality of elements and categorize them into a set of element types, many duplicated computations can be avoided to improve the entire performance of the software system.

Another knowledge source is learning systems. In engineering practice, many parameters are assumed for trials or obtained from previous experience. The proper design to certain cases does not guarantee its effectiveness in other cases. A continuing modification of design parameters is necessary. Equipped with the learning system, the proposed architecture can evaluate those results from each iteration of trials and compare them with early results. After several adjustments, the results of computation should converge to a desired range. The experience learned from one problem solving procedure can also be stored and used to support other similar cases thereafter.

The blackboard is divided into small portions (Figure 6). In horizontal direction, it is divided for each module to share. The data accessibility in one module from other
modules should be determined during module design process. In vertical direction, the blackboard is divided into two sections, the first half is for storing regular numerical data, and the other half is for controlling data and certain knowledge information.

There is one unified blackboard for all modules. This makes data management easier during computation. Data posted on the blackboard should correspond to the module hierarchy. For example, if two modules have inheritance relationships, the ancestor module's data are inherited to descendent module. If data in these two modules are separated on two blackboards, there will be an accessing problem. At least, the reference information of this inheritance relationship should be kept somewhere.

Applying OOP in managing data on this blackboard is desirable. Because OOP defines not only data of an object, but also methods of how to access data. With all data stored in the form of objects, no confusion or interference of data on the blackboard will occur. Only certain certified methods have accessibility to data on the blackboard. Using a database management system specially for the blackboard can be avoided. Therefore, the software system is more consistent in terms of structure.

We assume there is a trial-and-error process in numerical analysis for computer-aided engineering. Each iteration of computation creates a set of new data. We call one iteration as one version of design process. In a multiple version design process, a huge amount of data are created. Some data in one version maybe the same as in another version, but other data are different. In correspondent to such situation, the blackboard is designed as a dynamic database, one version for each modification (deletion, addition, renewing, etc.) of key parameters. This design is similar to the dynamic concept of the framework behind Minsky's original ideas [Minsky, 1975]. With detailed information of all versions stored on their correspondent blackboards, the evaluation and learning tasks become easier.
3.4 Implementation of Blackboard Architecture for Numerical Analysis

The above section outlines the principles of the new smart blackboard architecture for numerical analysis based on the conventional blackboard architecture. This section discusses several detail implementation issues regarding BANA. Considering the diversity of problems in computer-aided engineering, this discussion does not intend to give a complete and definite answer. Some details are further discussed in the subsequent chapters.

3.4.1 Object-Oriented Programming Paradigm

The architecture proposed above is a complicated structure for software developers. However, much attention should be paid on the major components, including modules, blackboard and global-local level relationships. Implementation of minor components are flexible. We recommend the object-oriented programming paradigm as a unified developing tool for BANA. This recommendation is based on OOP's four characteristics, modularity, inheritance, encapsulation and data abstraction.

**Modularity:**

In our architecture, each different knowledge source and its related controller is defined in one module in terms of object. Within each module, we have knowledge and actions. This matches the class structure of data and operations in object-oriented programming paradigm. In general, operations only work on data within the class scope. Assignments to software developers can be in the unit of one or more classes (modules). As long as the class interface is pre-defined among software engineers, individually developed classes are able to work together smoothly.

**Inheritance:**

We develop the concept of global-local level according to the data dependency among modules. This is true in many cases. For example, element stiffness computation
in finite element analysis depends on element geometry definition. Such dependency can be resolved by inheritance relationship provided by OOP. If we design the element geometry module at the global level, and element stiffness module at the local level, we can declare the inheritance relationship between element geometry module and element stiffness module. Element stiffness module can share the data and operations from element geometry module.

**Encapsulation:**

Encapsulation of data and operations in a class enhances the modularity of software systems. All related information and correspondent operations are protected from undesirable influence from other modules. This is specially beneficial to software debugging, modification and maintenance.

**Data Abstraction:**

Although engineering world is as diverse as the view from a kaleidoscope, there are still many similarities. Classes defined for software development is an abstraction from the real world. We categorize objects according to their similarities, and look for those behaviors and information commonly shared by all objects. Such abstraction work is necessary to simplify engineers' work. Otherwise we can not deal with tremendous number of tasks in the real world on time. Fortunately, many available industrial standards, computational mechanics, and design codes are all excellent abstraction resources for developing our software systems.

**OOP Class and BANA:**

The concept of OOP can be reviewed elsewhere, for example [Lee and Arora, 1991; Yu and Adeli, 1991]. In the OOP paradigm, we can present the concept of proposed architecture by a general class as shown in Figure 7. A module is represented by a class. Accessing to the blackboard is carried out by methods `input` and `output`. They are interfaces of one module to other modules. **Controller** is the commanding
Figure 7. Class representation of BANAmeth method of the class.
It sends the message to the knowledge sources to activate them. As described above, the knowledge sources can activate other modules which are objects residing in the definition of the method. Constructor is a special method in C++ object-oriented programming language [Stroustrup, 1991] to initialize an object from a class.

3.4.2 Group of Blackboards

Design objects are instantiated from the same design class with different groups of data values. We call them alternative design objects. The data used in design objects are varied, some are exactly same in all design objects, others are different. These alternative design objects compose a series of versions of the design. The purpose of the design process is to choose the most appropriate design object from all versions of alternative design objects. Generally we post one design object and its related data information (they can be either simple data or data objects) on one blackboard, in other words, one version one blackboard. For a set of design objects, we need a group of blackboards (Figure 8). An effective version management model for parallel blackboards in BANA is necessary. It is discussed in detail in Chapter VI.

3.4.3 Blackboard Organization

The blackboard in the proposed architecture is shared among all modules in one version. There are two types of blackboard: temporary and permanent. The temporary blackboard stores data for one version of design process. Those data with large volumes, such as element stiffness matrix, are stored on computer logic memory. At the end of design process of one version, only important data are stored into database to form permanent blackboard. The permanent blackboard is for evaluating results from different versions of design process, as in the learning system mentioned before.
3.4.4 Extended Entity-Relationship Model

With the increase of complexity of application problems, large quantity of information, mostly data, will be posted on the blackboard temporarily or permanently during computation. To efficiently organize these information, we apply techniques of database management systems into the proposed blackboard architecture.

The data entities in engineering problems are usually more complicated than those in business applications. They have both composition and inheritance relationships involved. The commonly used entity-relationship (ER) data model [Chen, 1976] is not appropriate for this purpose. An object-oriented data model based on extended entity-
relationship (EER) data model [Teorey et al., 1986] for numerical analysis is necessary for this purpose. The extended entity-relationship model allows composition and inheritance relationships among data attributes. It covers the data relationships in most engineering applications. The application of EER data model is further discussed in Chapter IV.

3.4.5 Knowledge Sources

We consider three types of knowledge sources: industrial standards, knowledge-based expert systems and learning systems. Most industrial standards are embedded in class method definitions based on problem solving algorithms.

Conventional computer-aided engineering systems usually have two separate developing paradigms for numerical analysis and knowledge-based systems, for example FORTRAN for the former and OPS5 for the later. This brings the interface problems of two different systems. In contrast, OOP provides an unique paradigm for both systems. We can develop the entire knowledge source in OOP paradigm [Adeli and Hung, 1990].

3.4.6 Concurrent Object-Oriented Programming

The advantage of developing proposed blackboard architecture for numerical analysis in object-oriented programming paradigm is the convenience of implementing concurrent programming. The software developed in OOP are highly structured in terms of modules. Each module can be executed as one process on a processor. The existence of blackboard provides communication tool for all processes. Although the real concurrent object-oriented programming language is still not available for this research, a simulation of concurrent programming can be achieved on an UNIX operating system platform. The following facilities in UNIX operating system can be used for simulation: new process generation, semaphores, pipes, named pipes, message queues, and shared
memory. The synchronization of module execution is achieved by using semaphores or message passing through pipes. Semaphore variables residing on the blackboard are examples of controlling data in BANA.

3.5 Application to Composite Laminate Analysis

In this section, we apply the algorithm of blackboard architecture for numerical analysis to a complex engineering problem: analysis of interlaminar stresses and strains in a composite laminate.

A composite laminate is composed of more than one layer of lamina in different stacking sequence (orientation and materials), as shown in Figure 9. It can carry either mechanical load, such as bending, moment, and extension, or thermal load or both. Under the external load, the stress within the laminate occurs. With the increase of stresses, matrix can no longer hold two adjacent laminae. This is the delamination of composite, a major cause of laminate failure.

Engineers have worked a long time to predict the interlaminar stress development between two adjacent layers. One of them is called global-local model [Pagano and Soni, 1983]. In this approach, the laminate is divided into two regions, one is the global region where the detailed stress and strain distribution is not of interest, and the other one is the local region where each lamina is analyzed separately in detail to find the interlaminar stresses and strains (Figure 10).

A finite element analysis software system has been developed for the global-local model [Chaturvedi et al., 1992]. The detail of the system is out of the scope of this chapter. The theory of global-local model is presented in Chapter II. We are only going to present the element stiffness matrix formulation of both global and local regions and the general structure of the system, from the view of BANA.
Figure 9. A laminate

Figure 10. The global-local model
The total potential energy in one element can be formulated in the following equation:

\[
\pi_{\text{element}} = \pi_{\text{global}} + \sum_{i=1}^{\text{local}} \pi_{i,\text{local}} = \frac{1}{2} \begin{bmatrix} K_1 + K_2 + K_3 + K_4 + K_5 \end{bmatrix} \begin{bmatrix} u_c \end{bmatrix} + \sum_{i=1}^{\text{local}} \left( R_1 + R_2 \right)
\]

(78)

where \( \pi_{\text{element}} \) is the total potential energy in one element, and it is the sum of that energy in global and local regions. Because local region has more than one layers, a summation of each layer's energy is necessary. Each matrix inside the bracket of Eq. (78) represents one entity to be added to the element stiffness matrix. Except their differences in formulation, those matrices all include the following computations: shape functions, Jacobian matrix, equations from FEA discretization, material stiffness and compliance matrices, etc.

To implement the above equation in finite element analysis, we use classes in our object-oriented finite element analysis class library. Classes included in the library are BOUCOND, ELEMENT, ELEMENT_SEQ, GAUSS, GENERAL_DATA, JACOB, NODE, MATERIAL, MATRIX, SHAPE, SHAPE_4, SHAPE_6 and VECTOR. For the composite laminate analysis, we need more specialized classes: GLOBAL_DATA, GLOBAL_ELEMENT, LOCAL_ELEMENT, NODE_GL, OBJECT_RECORD, OVM, STANDARD_TYPE, and VERSION_RECORD. The function of each class is listed briefly in the following.

- **BOUCOND**: boundary conditions
- **ELEMENT**: basic element, for further deriving special element
- **ELEMENT_SEQ**: element node sequence
- **GAUSS**: Gauss points for numerical computation
Some classes listed above are inherited from a general class (virtual class), such as class SHAPE_4 to SHAPE; some classes are part of another class's definition, such as JACOB and SHAPE_4 to GLOBAL_ELEMENT and LOCAL_ELEMENT. The advantage of applying OOP in FEA is that those standard concepts of FEA are represented by generalized classes and they are available for building up any FEA application software systems. For special FEA, only a limited amount of new specialized classes are necessary, and most of them can be derived from existing generalized classes. A great programming effort is saved through code reusing.

Figure 11 illustrates the BANA structure for computing element stiffness matrix. The element level controller and knowledge sources are in the global level with access to both blackboard and database. If the necessary data for computing stiffness matrix are already in the database, they are retrieved from the database and written on the
blackboard. Modules at local level have no direct access to database. They can only read and write data on the blackboard. One exception is the final result of stiffness matrix. The stiffness matrix is written on a special database instead of occupying blackboard space. Because stiffness matrix usually takes large amount of memory and won't be kept after linear equations are solved.

The concurrent computation can be achieved by assigning element controller to one processor, and local level controllers for both global region and local region to other processors. This assignment can be further carried out to computing matrix K1 through K6 in global region and R1 to R2 in local region. However, to avoid repeatedly compute shape functions, Jacobian matrices, material stiffness and compliance matrices, and so on, it is desirable to compute all of them in the element level and share in all processes on the blackboard. This posting is implemented by using shared memory on the computer.

Figure 12 shows the module of entire program controlling system. It has five primary knowledge sources, Preprocessor, Evaluation, Stiffness Matrix [K], Solver, and Postprocessor. The names of modules explain their function explicitly. The key modules are Evaluation, Stiffness Matrix [K] and Solver.

The purpose of proposing BANA is to provide a smart numerical analysis algorithm in OOP paradigm. Getting the final results is the goal of all computations. However, how to get them more efficiently and accurately is a challenge to engineers. We intend to apply as much heuristics as possible in the process. The knowledge for this application are related to finite element analysis, global-local model, equation solver, and machine learning (Figure 13). In each knowledge source, there are more than one piece of knowledge (rule). For example, in the Machine Learning Knowledge Source, we keep track of the blackboard version, and compare results from different versions. The record of blackboard version is extremely important in finite element analysis. Each new trial
Figure 11. Element stiffness matrix computation controlling system
Figure 12. Program controlling system
out does not guarantee an improved solution. All solutions are comparatively accurate in
certain regions of the mesh. For different purposes, we need appropriate approximation
methods.

Many equation solvers have been introduced for FEA to solve linear, non-linear,
symmetric, and unsymmetric equations. Frontal method is one of them. However,
programming a solver in frontal method is not easy in traditional programming. With
OOP it is very easy to do so. Because it is object-oriented in nature [Miller, 1989].

The whole computing process is taking place iteratively. These iterations can take
place concurrently or sequentially, depends on the requirements of the problem. Within
one iteration, certain parameters and conditions are assumed. If the results do not go to
the right direction, modification is required. This can be either determined by the
heuristics (convergence test) within the controller (it can be another knowledge source if
the structure or knowledge involved is too complicated), or by engineers who are
monitoring the computation. In other words, the whole process can be directed by users
interactively. We believe a system with a combination of human and machine knowledge
bases is an appropriate approach to the complex engineering problems. Computer can
provide a powerful memory in case study, but human experts are good at logic reasoning.

3.6 Closure

In this chapter, we presented a blackboard architecture for numerical analysis.
This architecture consists of blackboard architecture, extended entity-relationship model,
finite element analysis, knowledge-based systems, learning systems, and concurrent
programming under one unified object-oriented programming paradigm. One complex
engineering software system has been developed on the basis of this architecture. It
shows that the object-oriented programming can not only be applied to general software
engineering, but also to the numerical analysis in computer-aided engineering.
Figure 13. Hierarchy of knowledge sources in Evaluation KS
CHAPTER IV

OBJECT-ORIENTED FINITE ELEMENT ANALYSIS USING AN EER MODEL

4.1 Introduction

After more than 30 years of research and development finite element analysis is now used widely for analysis of numerous physical and engineering problems. With the increase in the size and complexity of finite element software systems the traditional waterfall algorithm-driven structured programming approach and the FORTRAN-77 language are no longer adequate and new software architecture and programming paradigms need to be created.

The object-oriented programming (OOP) paradigm with its characteristics of abstraction, inheritance, modularity, and encapsulation of data and operations provides a highly flexible and modular programming environment for analysis and design of complex software systems [Booch, 1991].

Forde and Stiemer [1989] show how OOP can be used to improve the design and implementation of interactive engineering software systems such as finite element analysis programs. Adeli and Hung [1990] present an object-oriented model for processing earthquake engineering knowledge. Knowledge representation is through a combination of frames and scripts. Two general purpose object classes provide support for developing and managing frame and script knowledge bases. Yu and Adeli [1991] present a CAD model using the OOP paradigm and a blackboard architecture for
management of input/output data as well as intermediate data created during a consultation with the CAD system.

4.2 Finite Element Analysis

Many mathematical models of physical and engineering problems can be described by differential equations. The finite element analysis is a numerical approach for the solution of these differential equations. In general, FEA involves seven steps [Bickford, 1990]: 1) Discretization, 2) Interpolation, 3) Element formulation, 4) Assembly of elements, 5) Applications of boundary conditions, 6) Solution of unknown variables (e.g., nodal displacement degrees of freedom), and 7) Computation of derived or secondary variables. The fundamental and most critical concepts are included in Steps 1-3; these are the main issues to be addressed in this paper. Steps 4-6 involve using a standard equation solver such as the frontal method [Bathe, 1982] for mainframe computers or out-of-core skyline method [Cheng, 1989] for personal computers. An object-oriented linear equation solving system has been developed and will be described in Chapter VIII.

In a linear FEA, the following equation is solved:

\[ K U = F \]  \hspace{1cm} (79)

where \( K \) is the global stiffness matrix, \( U \) is the generalized displacement vector, and \( F \) is the generalized force vector. The following four procedures are used in finite element modeling in generating global stiffness matrix \( K \) of Eq. (79): discretization, interpolation, mapping, and numerical integration.
4.3 Object-Oriented Enhanced Entity-Relationship Model

Data involved in a computer-aided engineering (CAE) software system are intensive and complex, and often must be shared by its subsystems. This necessitates the integration of a database management system (DBMS) into the CAE system. However, this integration is not straightforward because DBMS and CAE software systems are usually based on two different programming paradigms. In other words, there is a mismatch between the database manipulation language (DML) and the general purpose programming languages for engineering applications. Moreover, conventional DBMS, such as the relational data model cannot implement complex relationships among entities, such as "part-of" relationship. In this research, we integrate DBMS techniques into finite element analysis in one unified object-oriented programming paradigm to overcome the aforementioned difficulty. We first present a conceptual enhanced entity relationship model, and then implement it in the OOP environment.

An entity relationship (ER) model is the data model representing entities and relationships among them. Each entity corresponds to an object in the real-world. A set of similar entities defines an entity type which shares the same attributes. For example, nodes 1 to 4 in the finite element analysis (Figure 14) have similarity. Thus, the NODE entity type is defined to represent them. It has six attributes (Figure 15): Num (identification number), Stat (status), Disp (displacement), XYZ (coordinate), DOF (degrees of freedom) and Dim (coordinate dimension). The underline on Num indicates it is a key attribute whose value can distinguish an individual entity. Two attributes XYZ and Disp in NODE entity are composite attributes, because they are further divided into three coordinates of the node x, y, and z, and the nodal displacements in the x, y, and z directions u, v, and w.
Figure 14. A four-node element

Figure 15. Entity NODE in the ER model for general FEA
EER is an extension of the popular entity relationship model with superclass/subclass and generalization/specialization features [Elmasri and Navathe, 1989]. Therefore, it can represent complex data models more effectively than the ER data model. For example, a node in a general purpose FEA has three coordinates and three displacements (Figure 15) as discussed earlier. But, a node in the global-local modeling of composite structures, to be discussed in a subsequent section, has thirteen generalized displacements at each node instead of three. If the ER data model is used to represent this entity, a completely new entity type has to be defined, with array \[ u[13] \] replacing \[ u, v, \] and \[ w \] in addition to attributes \[ x, y, \] and \[ z \] (Figure 16). However, this shortcoming of the ER data model for a composite laminate node can be overcome by the EER data model, since it allows attribute inheritance between entity types which have superclass/subclass relationship. In the EER data model, only components \[ u[1], u[2], ..., u[13] \] of the attribute \[ Disp \] are re-defined in the new entity type, i.e. the new entity NODE_GL needs only \[ u[1], u[2], ..., u[13] \] as its attributes (Figure 17).

![Figure 16. Entity NODE in the ER model for composite laminate analysis](image-url)
Traditional database management systems are based on implementation of record-oriented data models, such as relational, hierarchical, and network models. These data models are developed primarily with business applications in mind. They are not entirely satisfactory for engineering applications, because engineering problems often require complex data and data relationships. Using the object-oriented data model, we can circumvent the shortcomings of the traditional data model.

Object-oriented data model is based on the principles of object-oriented programming, and supports all traditional data model functions at the same time. Object-oriented programming is a tool for software development. The core of OOP is the class, which is the abstraction of real-world objects. Class encapsulates both data and behaviors of objects in one module. The interactions among classes are predetermined by methods, so that the sources of many potential bugs appearing in conventional structured programming are eliminated. The inheritance relationship between the superclass and its subclasses promotes the code reusability and class extensibility in software development. A detailed description of basic OOP concepts and its advantages over the conventional structured programming can be found in Powell and Abdalla [1989], Fenves [1990], Lee and Arora [1991], and Yu and Adeli [1991].

OOP has the following features and advantages for DBMS development:

1) Objects correspond to real-world entities directly,

2) Possible redefinition of data and operations provides extendibility to entity types,

3) An entity schema can be defined in a class,

4) Complex data types can be represented by data abstraction in a class,

5) The inheritance relationship between superclass and subclasses allows defining specialized entity type from a generalized entity type,
6) Encapsulation of data and operations enhances the modularity of entities, and
7) There is no limitation on the number of attributes associated with one entity when it is implemented as a class.

Two classes representing the ER and EER data models corresponding to entities in Figures 19 and 20 are presented in Figures 21 and 22, respectively. In the ER model, an independent class must be defined for each new application. In contrast, in the EER data model as long as a class \textit{NODE} representing the general node entity is pre-defined in the class library, only a subclass with additional information needs to be created. For example, we define the class \textit{NODE\_GL} which includes additional nodal displacements for composite laminate analysis as a subclass of class \textit{NODE}. Since the subclass \textit{NODE\_GL} inherits from the superclass \textit{NODE}, it is possible to access the coordinates \(x\), \(y\), and \(z\) from the superclass. It is not necessary to rewrite the method \textit{get\_xyz()} in the subclass. To incorporate the thirteen unknown displacements, a new method \textit{get\_u()} is defined in the subclass \textit{NODE\_GL} to overwrite the no longer useful method \textit{get\_uvw()}. Comparing the two classes of \textit{NODE\_GL} in Figures 21 and 22, the EER model in the OOP environment demonstrates the advantages of class extensibility and code reusability.

![Diagram of NODE\_GL entity in EER model](image)

Figure 17. Entity NODE\_GL in the EER data model for composite laminate analysis
Figure 18. Class for Entity NODE in Figure 16

Figure 19. Class for Entity NODE_GL in Figure 17
While database management systems reside permanently within the computer's main memory, the instance of a class in an object-oriented programming environment resides on the main memory temporarily, only during the execution of the program. This can be a problem when information must be stored and made available throughout the execution of the program.

We employ two methods to overcome this inherent limitation. The first method is to have a database reside permanently on the computer's secondary storage, so that whenever access to the database is necessary, an instance can be created to perform that task. This method can be accomplished by the input/output library provided by the C++ language. For example, if we open a file through an instance `ofstream` of class `ofstream` (output file stream, a standard class of C++ I/O library), an instance `node` of class `NODE` can be stored in the database as:

```
output_file.write((char *)&node, sizeof(NODE));
```

To retrieve the data from the database, we can simply create an instance `ifstream` of class `ifstream` (input file stream, another standard class of C++ I/O library) and read the node data as:

```
input_file.read((char *)&node, sizeof(NODE));
```

The second method is to create an object within the program on a global level, i.e. in the main program, such that it can be accessed throughout the program's execution period.

To store specific information regarding nodes, elements, the stiffness matrix of each element, and so on, the first method is used, because this type of information requires a large amount of memory. Once that information is no longer required, the permanent memory can be deallocated. For general information which does not require
large memory, such as the number of nodes, number of elements, and element types in
the analysis, the second method is preferred and consequently used in OOFEA.

4.4 Class Library of Object-Oriented Finite Element Analysis

In this section, the classes necessary for object-oriented finite element analysis
are presented. We create 11 classes in the OOFEA class library. They are: VECTOR,
MATRIX, GENERAL_DATA, SHAPE, SHAPE_4, SHAPE_6, JACOB, NODE,
GAUSS, ELEMENT_SEQ, and ELEMENT. Only head files of classes are presented to
illustrate class specification, class organization and class relationships. The
implementation of methods is not presented for the sake of brevity.

There are various ways to represent a class. Graphic notations, such as Figures 21
and 22, are appropriate for illustrating relationships among classes, while class cards,
such as Figure 20, are more appropriate for listing the detailed contents of classes. All
class head files are presented in the class card form in Figures 23 through 33.

The original class card as proposed by Wirfs-Brock et al. [1990] is a simple index
card without any format. The purpose of such a class card is to record class designer's
ideas, primary methods, data, etc. It makes the class card easy to read, write, and delete
information during the process of designing classes. We extend the concept of class card
to document a class content with six components: Class Name, Super Class, Composed
Class, Data, Method, and Non-member Function. The Composed Class lists those
classes used in the construction of the new class, corresponding to the "part-of"
relationship in the class hierarchy. The Non-member Function lists operations that do not
belong to the class definition, but have strong relationships with the class, for example, an
operation between two instances of the same class.
Class **VECTOR**

Complex engineering problems require manipulation of vectors and matrices. OOP languages such as C++ do not provide such facilities directly. Class **VECTOR** has been created for performing vector manipulations (Figure 20), such as numerical and logical manipulation of vectors (addition, subtraction, multiplication, division by a scalar number, *etc.*), display of vectors, creating unit vectors, inputting of vector entities from the keyboard, and so on. It has six constructors (for initialization of an object of a class), one destructor (for de-allocation the memory no longer needed for an object), seventeen public methods (methods that can be accessed by operations of other classes), three private methods (methods that can only be accessed by the class members) and six other non-member functions. The basic data of the class are dimension of vector ($D_{im}$), an array of double precision float numbers ($Array$), and an integer number ($form$) to indicate the vector type (row or column for displaying/ printing purposes).

With different constructors, the instances of **VECTOR** can be created in response to different requirements. For example, **VECTOR**(const **VECTOR**&) creates a new instance from an old instance and **VECTOR**(int, double ...) creates an instance with entities provided by a list of variables. Those user-defined overloaded operations (two or more operations using the same name) give the user the flexibility of manipulating vectors in a way similar to processing scalar numbers. For example, the method **VECTOR**& operator += (const **VECTOR**&) allows a vector $V_1$ (1.1, 1.2) to be added to another vector $V_2$ (2.1, 2.2):

$$VECTOR V_1(2, 1.1, 1.2), V_2(2, 2.1, 2.2);$$
$$V_2 += V_1;$$

This operation yields values of 3.2 and 3.4 for the variables of $V_2$. 
<table>
<thead>
<tr>
<th>Class Name</th>
<th>VECTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Class</td>
<td></td>
</tr>
<tr>
<td>Composed Class</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>int Dim; double *Array; int form;</td>
</tr>
<tr>
<td>Method</td>
<td></td>
</tr>
<tr>
<td>private:</td>
<td></td>
</tr>
<tr>
<td>VECTOR&amp; SetUp(int);</td>
<td></td>
</tr>
<tr>
<td>VECTOR&amp; Keyln();</td>
<td></td>
</tr>
<tr>
<td>VECTOR&amp; Copy(const VECTOR&amp;, int);</td>
<td></td>
</tr>
<tr>
<td>public:</td>
<td></td>
</tr>
<tr>
<td>VECTOR();</td>
<td></td>
</tr>
<tr>
<td>VECTOR(int);</td>
<td></td>
</tr>
<tr>
<td>VECTOR(int, double ...);</td>
<td></td>
</tr>
<tr>
<td>VECTOR(int, double *);</td>
<td></td>
</tr>
<tr>
<td>VECTOR(const VECTOR&amp;);</td>
<td></td>
</tr>
<tr>
<td>VECTOR(const VECTOR&amp;, int);</td>
<td></td>
</tr>
<tr>
<td>-VECTOR();</td>
<td></td>
</tr>
<tr>
<td>VECTOR&amp; MakeZero();</td>
<td></td>
</tr>
<tr>
<td>VECTOR&amp; MakeUnit();</td>
<td></td>
</tr>
<tr>
<td>VECTOR&amp; AllEntries(const double&amp;);</td>
<td></td>
</tr>
<tr>
<td>int DimIndex();</td>
<td></td>
</tr>
<tr>
<td>int get_form() const;</td>
<td></td>
</tr>
<tr>
<td>void PrintForm();</td>
<td></td>
</tr>
<tr>
<td>void FtoV(const double *, int);</td>
<td></td>
</tr>
<tr>
<td>void VtoF(double *, int);</td>
<td></td>
</tr>
<tr>
<td>double&amp; operator<a href="int"></a> const;</td>
<td></td>
</tr>
<tr>
<td>VECTOR&amp; operator +=(const VECTOR&amp;);</td>
<td></td>
</tr>
<tr>
<td>VECTOR&amp; operator -=(const VECTOR&amp;);</td>
<td></td>
</tr>
<tr>
<td>VECTOR&amp; operator *=(const VECTOR&amp;);</td>
<td></td>
</tr>
<tr>
<td>VECTOR&amp; operator =(const VECTOR&amp;);</td>
<td></td>
</tr>
<tr>
<td>friend VECTOR operator + (const VECTOR&amp;, const VECTOR&amp;);</td>
<td></td>
</tr>
<tr>
<td>friend VECTOR operator - (const VECTOR&amp;, const VECTOR&amp;);</td>
<td></td>
</tr>
<tr>
<td>friend double operator * (const VECTOR&amp;, const VECTOR&amp;);</td>
<td></td>
</tr>
<tr>
<td>friend ostream&amp; operator &lt;&lt; (ostream&amp;, const VECTOR&amp;);</td>
<td></td>
</tr>
</tbody>
</table>

| Non-member Function | int operator == (const VECTOR&, const VECTOR&); |
|                     | int operator != (const VECTOR&, const VECTOR&); |
|                     | VECTOR& operator * (const double&, const VECTOR&); |
|                     | VECTOR& operator += (const VECTOR&, const VECTOR&); |
|                     | VECTOR& operator -= (const VECTOR&, const VECTOR&); |
|                     | double operator *= (const VECTOR&, const VECTOR&); |
Some vector operations are performed between two \textit{VECTOR} instances, where the result belongs to a third \textit{VECTOR} instance. One way of designing such methods is creating a non-member function, as in the class \textit{VECTOR}. Another way is declaring two instances as friends (allowing private members to be accessed by classes outside their inheritance hierarchy) to another instance, as in the class \textit{MATRIX}, to be discussed shortly. The former method needs a third instance before the operations performed, while the latter method creates an instance during the operation. For the convenience of users, both alternatives are provided in our OOFEA.

\textsc{Fortran} language is still the most popular language for engineering applications, and thousands of existing codes are programmed in this language. Classes \textit{VECTOR} and \textit{MATRIX} collectively provide methods for converting one- and two-dimensional arrays defined in \textsc{fortran} to the C++ language and vice versa. With these methods, subroutines coded in \textsc{fortran} can be linked to an OOFEA application program without re-coding them in the C++ language. We will show the application of these methods by presenting an example in the next section.

Class \textit{MATRIX}

Matrix operations are more involved than vector operations. Class \textit{MATRIX} is created for performing matrix operations, such as addition, subtraction, multiplication, displaying the matrix, and so on (Figure 21). It has twenty-six public methods, three private methods, and seven non-member functions. Two non-member functions are for computation of inverse and transpose of a square matrix. In addition to the aforementioned matrix operations, this matrix class can perform union operation of two matrices to form a new matrix. The elements of a matrix in the class \textit{MATRIX} are defined as an array of \textit{VECTOR} instances.
<table>
<thead>
<tr>
<th>Class Name</th>
<th>MATRIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Class</td>
<td>MATRIX</td>
</tr>
<tr>
<td>Composed Class</td>
<td>VECTOR</td>
</tr>
<tr>
<td>Data</td>
<td>int Rdim, Cdim, form; VECTOR *Row:</td>
</tr>
</tbody>
</table>

### Method

**private:**
- MATRIX& SetUpO;
- MATRIX& KeyInO;
- MATRIX& Copy(const MATRIX&);

**public:**
- MATRIX():
- MATRIX(int, int);
- MATRIX(const MATRIX&);
- MATRIX(int, int, double*);
- MATRIX(const MATRIX&, int, ~MATRIX(); int, int, int);
- VECTOR& operator [ ] (int) const;
- MATRIX& operator =(const MATRIX&);
- MATRIX& operator += (const MATRIX&);
- MATRIX& operator -= (const MATRIX&);
- MATRIX& operator *= (const double&);
- friend MATRIX& operator + (const MATRIX&, const MATRIX&);
- friend MATRIX& operator - (const MATRIX&, const MATRIX&);
- friend MATRIX& operator * (const MATRIX&, const MATRIX&);
- friend MATRIX& operator / (const MATRIX&, const MATRIX&);
- friend MATRIX& operator *(const VECTOR&, const MATRIX&);
- friend MATRIX& operator *(const MATRIX&, const VECTOR&);
- friend MATRIX& operator ==(const MATRIX&, const MATRIX&);
- friend MATRIX& operator | (const MATRIX&, const MATRIX&);
- friend MATRIX& operator & (const MATRIX&, const MATRIX&);
- friend int operator == (const MATRIX&, const MATRIX&);
- friend ostream& operator << (ostream&, const MATRIX&);

### Non-member Function

- MATRIX inverse(const MATRIX&);
- MATRIX transpose(const MATRIX&);
- int operator== (const MATRIX&, const MATRIX&);
- int operator!= (const MATRIX&, const MATRIX&);
- MATRIX& operator + (const MATRIX&, const MATRIX&);
- MATRIX& operator - (const MATRIX&, const MATRIX&);
- MATRIX& operator * (const MATRIX&, const MATRIX&);
- MATRIX& operator / (const MATRIX&, const MATRIX&);
- MATRIX& operator | (const MATRIX&, const MATRIX&);
- MATRIX& operator & (const MATRIX&, const MATRIX&);

Figure 21. Class card of **MATRIX**
Class **GENERAL_DATA**

For any FEA application, there is a certain amount of general information for the entire analysis, such as total number of nodal points, elements, and so on. This information is handled by the instances of the class **GENERAL_DATA** (Figure 22). Other classes can request this information through the public methods in the class. However, for certain classes, the class **GENERAL_DATA** is used as a superclass to provide better service. This is an application of the inheritance characteristic of OOP, despite the fact that these two classes may not have any "kind-of" relationship. Class **ELEMENT** which will be discussed shortly is a subclass of class **GENERAL_DATA**.

Class **SHAPE**

After three decades of finite element research and development, many element types have been proposed, for example, Q4 (four-node quadrilateral element), T3 (three-node triangular element), T6 (six-node triangular element), etc., and more will be developed in the future. Each finite element has its own shape functions and derivatives of the shape functions. Therefore, we design a superclass **SHAPE** (Figure 23) which contains no detailed definition of shape functions and their derivatives. It allocates memory, operations, and data for its subclasses. The operation `virtual void definition(double ss, double tt)=0` is a special way in C++ to declare the class **SHAPE** as a virtual superclass. A virtual superclass, also called an abstract class, can not have instances of its own. It contains the names of some methods without containing their actual operations. These operations are defined in subclasses.
<table>
<thead>
<tr>
<th>Class Name</th>
<th>GENERAL_DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Class</td>
<td></td>
</tr>
<tr>
<td>Composed Class</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>int Num_nodal_point, Num_element, Num_bc_point,</td>
</tr>
<tr>
<td></td>
<td>Num_ele_node, Num_gauss, Node DOF, Total DOF,</td>
</tr>
<tr>
<td></td>
<td>Elem DOF, Num_ele_gauss, Dim_xyz;</td>
</tr>
<tr>
<td></td>
<td>char Title[30];</td>
</tr>
</tbody>
</table>

**Method**

```cpp
public:
    GENERAL_DATA();
    GENERAL_DATA(char*);
    GENERAL_DATA(const GENERAL_DATA&);
    ~GENERAL_DATA();

    void input_info();
    void check_info();
    int get_Num_nodal_point();
    int get_Num_element();
    int get_Num_bc_point();
    int get_Num_ele_node();
    int get_Num_gauss();
    int get_Node_DOF();
    int get_Total_DOF();
    int get_Num_ele_gauss();
    int get_Dim_xyz();
    void assign_Node_DOF(int);
```

Figure 22. Class card of `GENERAL_DATA`
<table>
<thead>
<tr>
<th>Class Name</th>
<th>SHAPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Class</td>
<td>VECTOR, MATRIX</td>
</tr>
<tr>
<td>Composed Class</td>
<td>VECTOR, MATRIX</td>
</tr>
<tr>
<td>Data</td>
<td>int node_num;</td>
</tr>
<tr>
<td></td>
<td>VECTOR shape;</td>
</tr>
<tr>
<td></td>
<td>MATRIX derive;</td>
</tr>
<tr>
<td>Method</td>
<td>public:</td>
</tr>
<tr>
<td></td>
<td>SHAPE(int);</td>
</tr>
<tr>
<td></td>
<td>~SHAPE();</td>
</tr>
<tr>
<td></td>
<td>VECTOR&amp; get_shape();</td>
</tr>
<tr>
<td></td>
<td>MATRIX&amp; get_derive();</td>
</tr>
<tr>
<td></td>
<td>int get_node();</td>
</tr>
<tr>
<td></td>
<td>virtual void definition(double ss, double tt) =0;</td>
</tr>
</tbody>
</table>

Figure 23. Class card of SHAPE

Classes SHAPE_4 and SHAPE_6

Classes SHAPE_4 (Figure 24) for Q4 element and SHAPE_6 (Figure 25) for T6 element have the same structure except for the content of the public method `definition(double, double)`. These two classes inherit from the superclass SHAPE. General users of OOFEA can define their own shape functions in these classes inheriting from the superclass SHAPE. Two features of OOP play an important role in the design of shape function classes. One is the code reusability, that is the aforementioned subclasses inherit the data (node_num, shape, derive) and certain methods [VECTOR& get_shape(), MATRIX& get_derive(), and int get_node()] from the superclass SHAPE. Another one is the overloading of methods, in which method `void definition(double, double)` is redefined in both subclasses SHAPE_4 and SHAPE_6.
### SHAPE_4

<table>
<thead>
<tr>
<th>Class Name</th>
<th>SHAPE_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Class</td>
<td>SHAPE</td>
</tr>
<tr>
<td>Composed Class</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>double s, t;</td>
</tr>
</tbody>
</table>

```cpp
public:
SHAPE_4(): SHAPE(4);
~SHAPE_4();
void definition(double, double);
```

Figure 24. Class card of SHAPE_4

###SHAPE_6

<table>
<thead>
<tr>
<th>Class Name</th>
<th>SHAPE_6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Class</td>
<td>SHAPE</td>
</tr>
<tr>
<td>Composed Class</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>double s, t;</td>
</tr>
</tbody>
</table>

```cpp
public:
SHAPE_6(): SHAPE(6);
~SHAPE_6();
void definition(double, double);
```

Figure 25. Class card of SHAPE_6
Class **JACOB**

The class **JACOB** (Figure 26) provides Cartesian derivatives of shape functions and determinant of Jacobian matrix for the coordinate transformation. The global coordinates of nodes and derivatives of shape functions are needed in the computation. Since class **JACOB** takes derivatives of shape functions as one parameter in its definition method, it is independent of the type of shape function.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>JACOB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Class</td>
<td></td>
</tr>
<tr>
<td>Composed Class</td>
<td>MATRIX</td>
</tr>
<tr>
<td>Data</td>
<td>int node_num; double s, t, djacb; MATRIX elcod, derive, cartd, xjaci, xjacm;</td>
</tr>
<tr>
<td>Method</td>
<td>public: JACOB(int MATRIX&amp;); ~JACOB(); void definition (MATRIX&amp;); MATRIX&amp; get_cartd(); double get_djacb();</td>
</tr>
</tbody>
</table>

Figure 26. Class card of **JACOB**
Class **GAUSS**

Class **GAUSS** (Figure 27) provides the locations of Gaussian points and their weights. Its structure is simple, with one constructor, one destructor, one private definition method, and two public methods to provide the location and weights of the Gaussian points. The number of Gaussian points in an element is inputted when the object is created.

Class **NODE**

Class **NODE** has eight constructors, one destructor, fifteen public methods, two private methods, and one non-member function as shown in Figure 28. Its function is to provide nodal geometric information and generalized displacements for element stiffness matrix computation, and to act as a tool for accessing the node EER database. The latter function will be discussed later.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>GAUSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Class</td>
<td></td>
</tr>
<tr>
<td>Composed Class</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>int gp_num; double *Weight, *Position;</td>
</tr>
<tr>
<td>Method</td>
<td></td>
</tr>
<tr>
<td>public:</td>
<td></td>
</tr>
<tr>
<td>GAUSS(int); -GAUSS(); double *get_weight(); double *get_position(); int get_gp_num();</td>
<td></td>
</tr>
<tr>
<td>private:</td>
<td></td>
</tr>
<tr>
<td>void def_gauss();</td>
<td></td>
</tr>
</tbody>
</table>

Figure 27. Class card of **GAUSS**
<table>
<thead>
<tr>
<th>Class Name</th>
<th>NODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Class</td>
<td></td>
</tr>
<tr>
<td>Composed Class</td>
<td></td>
</tr>
</tbody>
</table>
| Data | double xyz[3], disp[3];  
int NodeNum, NumDim, NumDOF, Status; |

<table>
<thead>
<tr>
<th>Method</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>private:</td>
<td></td>
</tr>
<tr>
<td>void SetUp(int, int, int);</td>
<td></td>
</tr>
<tr>
<td>NODE&amp; Copy(const NODE&amp;);</td>
<td></td>
</tr>
<tr>
<td>public:</td>
<td></td>
</tr>
<tr>
<td>NODE();</td>
<td></td>
</tr>
<tr>
<td>NODE(int, int, int);</td>
<td></td>
</tr>
<tr>
<td>NODE(int, int, int, double*, int*);</td>
<td></td>
</tr>
<tr>
<td>NODE(int, int, int, int*, double ...);</td>
<td></td>
</tr>
<tr>
<td>NODE(int, int, int, double*, int ...);</td>
<td></td>
</tr>
<tr>
<td>NODE(const NODE&amp;);</td>
<td></td>
</tr>
<tr>
<td>NODE(const NODE&amp;, const NODE&amp;);</td>
<td></td>
</tr>
<tr>
<td>NODE(const NODE&amp;, const NODE&amp;, const int);</td>
<td></td>
</tr>
<tr>
<td>-NODE();</td>
<td></td>
</tr>
<tr>
<td>int get_NodeNum() const;</td>
<td></td>
</tr>
<tr>
<td>int get_NumDOF() const;</td>
<td></td>
</tr>
<tr>
<td>int get_NumDim() const;</td>
<td></td>
</tr>
<tr>
<td>int get_Status() const;</td>
<td></td>
</tr>
<tr>
<td>double* get_xyz() const;</td>
<td></td>
</tr>
<tr>
<td>double* get_disp() const;</td>
<td></td>
</tr>
<tr>
<td>void assign_val(double *);</td>
<td></td>
</tr>
<tr>
<td>void assign_val();</td>
<td></td>
</tr>
<tr>
<td>void assign_status(int);</td>
<td></td>
</tr>
<tr>
<td>void assign_Node_num(int);</td>
<td></td>
</tr>
<tr>
<td>void assign_Num_Dim(int);</td>
<td></td>
</tr>
<tr>
<td>void assign_disp(const double*);</td>
<td></td>
</tr>
<tr>
<td>friend ostream&amp; operator &lt;&lt; (ostream&amp;, const NODE&amp;);</td>
<td></td>
</tr>
<tr>
<td>NODE&amp; operator +=(const NODE&amp;);</td>
<td></td>
</tr>
<tr>
<td>NODE&amp; operator -==(const NODE&amp;);</td>
<td></td>
</tr>
</tbody>
</table>

| Non-member Function | NODE operator + (const NODE&, const NODE&); |

Figure 28. Class card of NODE
In the previous section, we discussed the role of EER data model in the class NODE in detail. The class NODE has additional functions beyond storing and providing nodal geometry and the generalized displacement information. The automatic generation of nodes is also included in this class by designing a specific constructor and related methods.

First, we create an incremental node object (an abstract node which contains information about intervals of nodal sequence numbering and global coordinates) by the constructor NODE(const NODE& N1, const NODE& N2, const int num) to illustrate the polymorphism of OOP (the capability to use the same name for different functions). This object provides incremental information for uniformly-spaced nodes between two nodes N1 and N2. Then, new nodes can be created by simply adding the "incremental node" to the original node similar to an arithmetic addition. For example, consider two nodes node_1 at (0, 0, 0) and node_9 at (8, 8, 0), and suppose there are three additional nodes located evenly along the straight line connecting node_1 to node_9. To begin with, an "incremental node" Dnode is created:

\[
\text{NODE Dnode(node_1, node_9, 3);}
\]

The increments of coordinates (x, y, z) and node numbers are (2, 2, 0) and 2, respectively. Therefore, the intermediate nodes are created as follows:

\[
\begin{align*}
\text{NODE node_3} &= \text{node}_1 + \text{Dnode}; \\
\text{NODE node_5} &= \text{node}_3 + \text{Dnode}; \\
\text{NODE node_7} &= \text{node}_5 + \text{Dnode};
\end{align*}
\]

Uniformly placed nodes and elements do not have to be the only option in FEA. Special elements are necessary for particular problems. To generate such nodes and elements, the heuristics of experts can be applied to the code generation process. In other
words, the techniques of knowledge engineering [Adeli, 1990a and 1990b] [Adeli and Balasubramanyam, 1988] can play a useful role in this area. The class NODE created in the OOP environment can be extended to include the heuristics or knowledge bases for creating meshes of irregular geometry.

Class ELEMENT_SEQ

For each element, we need information about its node sequence. From the node sequence we can retrieve the coordinates of each node from the node EER database. We do not include the operation of obtaining node sequence within the class ELEMENT; rather, we create an individual class ELEMENT_SEQ (Figure 29) to perform this job. The reasons are as follows:

1) Class ELEMENT is the core of OOFEA, it already has many functions to perform;
2) The information of node sequence is generally inputted at the beginning of the analysis, and stored in the EER database;
3) Element stiffness matrix computation is performed after all the data are available, not at the time of inputting data; and
4) Class ELEMENT allocates a large amount of memory for the element stiffness matrix (for example, the global-local model of composite laminate analysis requires a 136x136 stiffness matrix for a four-layer composite laminate) and needs substantial processing time to create the element stiffness matrix; therefore, we create instances only for different types of elements to avoid computing stiffness matrices of all elements repetitiously and unnecessarily.

The structure of class ELEMENT_SEQ (Figure 29) is quite similar to that of class NODE. It has the function of generating elements automatically if elements are
evenly spaced in one direction. The related methods are constructor `ELEMENT_SEQ(const ELEMENT_SEQ&, const ELEMENT_SEQ&, const int)` for generating object of "incremental element" and method `ELEMENT_SEQ& operator += (const ELEMENT&)` for creating a new object based on the old object and the "incremental element". The main distinction between the two classes is that class `ELEMENT_SEQ` has an integer number array for recording the sequence of nodes within an element instead of double precision float number array for storing the nodal coordinates in the class `NODE`. The techniques of DBMS are applied in the definition of `ELEMENT_SEQ` class. One aim of this class is to determine which element type this element belongs to. The geometry of the element is one major consideration. The attribute `Seq/B` (node series) in element sequence entity stores the key attribute of the node entity. Two entities are related together.

Class `ELEMENT`  

Class `ELEMENT` (Figure 30) is a virtual superclass to different kinds of element that may exist in an application (Figure 31), for example, the elements in the global and local regions of composite laminate analysis, the element in a heat transfer problem, and so on. At the same time, for the convenience of accessing the general FEA information, we declare it as a subclass of class `GENERAL_DATA`. By this arrangement, the designer of `ELEMENT`'s subclasses does not have to be concerned about how the general data is provided during the initialization of the superclass `ELEMENT`.

We declare the class `GAUSS` as another superclass of the class `ELEMENT` because the element computation is characterized by the Gauss-Legendre method, and therefore, the "kind-of" inheritance relationship exists between the two classes. This arrangement creates a multiple-inheritance relationship between the class `ELEMENT` with classes `GENERAL_DATA` and `GAUSS`. 
<table>
<thead>
<tr>
<th><strong>Class Name</strong></th>
<th>ELEMENT_SEQ</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Super Class</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Composed Class</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Data</strong></td>
<td>int Seq[8], Ele_Num, Num_Node, Status;</td>
</tr>
<tr>
<td><strong>Method</strong></td>
<td></td>
</tr>
<tr>
<td>public:</td>
<td></td>
</tr>
<tr>
<td>ELEMENT_SEQ();</td>
<td></td>
</tr>
<tr>
<td>ELEMENT_SEQ(int, int);</td>
<td></td>
</tr>
<tr>
<td>ELEMENT_SEQ(int, int, int*);</td>
<td></td>
</tr>
<tr>
<td>ELEMENT_SEQ(int, int ...);</td>
<td></td>
</tr>
<tr>
<td>ELEMENT_SEQ(const ELEMENT_SEQ&amp;);</td>
<td></td>
</tr>
<tr>
<td>ELEMENT_SEQ(const ELEMENT_SEQ&amp;, const ELEMENT_SEQ&amp;, const int);</td>
<td></td>
</tr>
<tr>
<td>~ELEMENT_SEQ(void);</td>
<td></td>
</tr>
<tr>
<td>void SetUp(int, int);</td>
<td></td>
</tr>
<tr>
<td>int get_Ele_Num(void) const;</td>
<td></td>
</tr>
<tr>
<td>int get_Num_Node(void) const;</td>
<td></td>
</tr>
<tr>
<td>int get_Status(void) const;</td>
<td></td>
</tr>
<tr>
<td>int* get_Seq(void) const;</td>
<td></td>
</tr>
<tr>
<td>void assign_Seq(void);</td>
<td></td>
</tr>
<tr>
<td>void assign_Seq(int*);</td>
<td></td>
</tr>
<tr>
<td>void assign_Ele_Num(int);</td>
<td></td>
</tr>
<tr>
<td>void assign_Status(int);</td>
<td></td>
</tr>
<tr>
<td>friend ostream&amp; operator &lt;&lt; (ostream&amp;, const ELEMENT_SEQ&amp;);</td>
<td></td>
</tr>
<tr>
<td>ELEMENT_SEQ&amp; Copy(const ELEMENT_SEQ&amp;);</td>
<td></td>
</tr>
<tr>
<td>ELEMENT_SEQ&amp; operator += (const ELEMENT_SEQ&amp;);</td>
<td></td>
</tr>
<tr>
<td>ELEMENT_SEQ&amp; operator = (const ELEMENT_SEQ&amp;);</td>
<td></td>
</tr>
<tr>
<td><strong>Non-member Function</strong></td>
<td>ELEMENT_SEQ operator + (const ELEMENT_SEQ&amp;, const ELEMENT_SEQ&amp;);</td>
</tr>
</tbody>
</table>

Figure 29. Class card of **ELEMENT_SEQ**
| Class Name | ELEMENT |
| Super Class | GENERAL_DATA, GAUSS |
| Composed Class | VECTOR, MATRIX, ELEMENT_SEQ |
| Data | int *Lnods, Ele_Num, Num_node, Num_total_point, Dim_xyz, Num_gauss, Num_DOF, Ele_Type, Num_Element; double Dvolu; VECTOR Shape: |
| Method | MATRIX Stiff, Gpcod, Derive, Elcod, Cartd; |
| public: | ELEMENT(int, int, GENERAL_DATA&); ~ELEMENT(); |
| void def_Inods(); | virtual void definition()=0; virtual void def_Ele_coord()=0; |
| MATRIX& get_Stiff(); int* get_Inods() const; | int get_Elcod() const; |

Figure 30. Class card of ELEMENT
Figure 31. Class hierarchy of ELEMENT

Legend:

A: regular class
B: virtual class
Class B inherits from Class A
4.5 Application of OOFEA to Composite Laminate Interlaminar Stress Analysis

In this section, we apply the object-oriented finite element analysis class library to create the element stiffness matrix for composite laminate interlaminar stress analysis.

A composite laminate is composed of several layers of thin epoxy-glass-like sheets, called lamina (Figure 32). The material between the laminae is called matrix, and is used to hold the laminae together. The strength of the laminate is primarily determined by the fibers within the laminae. The unique composition of a given composite laminate determines its performance under external (physical or thermal) loads. The separation of adjacent laminae, called delamination, is the major cause of laminate failure used in aerospace structures. We use the global-local modeling approach [Pagano and Soni, 1983] for analyzing the stresses and strains in composite laminates (Figure 33). In this approach, the laminate is divided into two regions, one is called the global region where detailed stress and strain distribution is not of interest, and the other is the local region where each lamina is analyzed separately in detail to find the interlaminar stresses and strains.

The global region stiffness matrix can be formulated through minimization of the total potential energy of the system. The equation for the stiffness matrix of the global region is the integral term inside the bracket in the Eq. (80) [Chaturvedi, et al., 1992]:

\[
\begin{align*}
\pi_g &= \frac{1}{2} \left( \mathbf{u}_r \right)^T \int_{\Omega_{\text{global}}} \left[ \begin{array}{c}
\frac{h}{2} L_{12N}^T \mathbf{C}_{12N} L_{12N} + \frac{h}{2} L_{12N}^T \mathbf{C}_{212} L_{12N} \\
\frac{h}{2} L_{34N}^T \mathbf{C}_{34N} L_{34N} + \frac{h}{2} L_{34N}^T \mathbf{C}_{234} L_{34N} \\
\frac{h}{2} L_{56N}^T \mathbf{C}_{56N} L_{56N} + \frac{h}{2} L_{56N}^T \mathbf{C}_{256} L_{56N}
\end{array} \right] \ dA \mathbf{u}_r \\
\end{align*}
\]
where $\pi_g$ is the total potential energy in the global region, $\overline{C}_1$ and $\overline{C}_2$ are part of transformed laminate material stiffness matrix, $h$ is the height of the global region. $A_{global}$ is the area of the global region, $\mathbf{L}_m$ and $\overline{\mathbf{L}}_m$ are vectors containing Cartesian derivatives of shape functions (see [Chaturvedi et al., 1992] for their definitions), and $u_e$ is the displacement vector of the element.

To implement Eq. (80) using the OOFEA class library, we only need to create three more classes for the global region. They are called $\mathbf{NOD}_{-}GL$, $\mathbf{MATERIAL}$, and $\mathbf{GLOBAL}_{-}\mathbf{ELEMENT}$.
Figure 33. Global-local model

Figure 34. Thirteen unknown variables at each node in the global region

u: displacement in x direction (in.), p: normal stress in z direction (psi),
v: displacement in y direction (in.), s: shear stress in yz direction (psi),
w: displacement in z direction (in.), t: shear stress in xz direction (psi).
w_o: displacement in z direction at mid-plane of lamina (in.).
subscript b and t: bottom and top of region,
Class \textit{NODE\_GL}

As shown in Figure 34, there are thirteen degrees of freedom in the global region instead of three defined in the class \textit{NODE}. The primary purpose of the class \textit{NODE\_GL} is to add the additional unknown variables to each node. The two methods and one non-member function defined in the class \textit{NODE\_GL} (Figure 35) act as communication tools to access methods in the class \textit{NODE} based on EER data model used for these two classes.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>NODE_GL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Class</td>
<td>NODE</td>
</tr>
<tr>
<td>Composed Class</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>double disp[13];</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>public:</td>
</tr>
<tr>
<td>NODE_GL();</td>
</tr>
<tr>
<td>NODE_GL(const NODE_GL&amp;);</td>
</tr>
<tr>
<td>NODE_GL(int i1, int i2, int i3) : NODE(i1,i2,i3);</td>
</tr>
<tr>
<td>NODE_GL(const NODE_GL&amp; N1, const NODE_GL&amp; N2, const int incre) : NODE(N1, N2, incre);</td>
</tr>
<tr>
<td>~NODE_GL();</td>
</tr>
<tr>
<td>NODE_GL&amp; Copy(const NODE_GL&amp;);</td>
</tr>
<tr>
<td>friend ostream&amp; operator &lt;&lt; (ostream&amp;, const NODE_GL&amp;);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Non-member Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE_GL operator + (const NODE_GL&amp;, const NODE_GL&amp;);</td>
</tr>
</tbody>
</table>

Figure 35. Class card of \textit{NODE\_GL}
Composite laminates have special material properties not present in other commonly used materials, such as steel and concrete. It is necessary to create a class for computation of the composite laminate compliance and stiffness matrices. Class MATERIAL serves this purpose (Figure 36). Fibers embedded in the laminate do not always coincide with the direction of external loads. Two methods are designed for defining transformation matrices for stiffness and compliance matrices.

### Class MATERIAL

Composite laminates have special material properties not present in other commonly used materials, such as steel and concrete. It is necessary to create a class for computation of the composite laminate compliance and stiffness matrices. Class MATERIAL serves this purpose (Figure 36). Fibers embedded in the laminate do not always coincide with the direction of external loads. Two methods are designed for defining transformation matrices for stiffness and compliance matrices.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>MATERIAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Class</td>
<td>MATRIX</td>
</tr>
<tr>
<td>Composed Class</td>
<td>MATRIX</td>
</tr>
<tr>
<td>Data</td>
<td>double E11, E22, E33, v12, v23, v13, v21, v32, v31, G12, G23, G13, angle, m, n, mm, mn; MATRIX S, C;</td>
</tr>
<tr>
<td>Method</td>
<td>public: MATERIAL(double*, int, int); ~MATERIAL(); MATRIX&amp; get_S(); MATRIX&amp; get_C();</td>
</tr>
<tr>
<td></td>
<td>protected: void def_S(); void def_C(); void def_Trans_S(MATRIX&amp;); void def_Trans_C(MATRIX&amp;);</td>
</tr>
</tbody>
</table>

Figure 36. Class card of MATERIAL
Class *GLOBAL_ELEMENT*

Class *GLOBAL_ELEMENT* (Figure 37) is for implementing Eq. (3) by assembling instances of those classes we discussed so far. It inherits from class *GENERAL_DATA*. The class relationship in designing class *GLOBAL_ELEMENT* is shown in Figure 38. Besides one superclass, this class uses instances of four classes directly as its components. Other classes are used indirectly. Material properties in terms of stiffness and compliance matrices are passed as parameters during initialization. In the global-local modeling of composite laminates, additional information is needed beyond what is provided by the class *GENERAL_DATA*. This information is inputted through the method `void add_info()`. The reason for redefining the method `void def_Ele_coord()` originally defined for the class *ELEMENT* for retrieving nodal coordinates is that we store nodal information via instance of class *NODE_GL*, which is defined specifically for the global-local modeling of composite structures.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>GLOBAL_ELEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Class</td>
<td>ELEMENT</td>
</tr>
<tr>
<td>Composed Class</td>
<td>MATRIX, JACOB, MATERIAL, NODE_GL</td>
</tr>
<tr>
<td>Data</td>
<td>double Hglobe, int Node_DOF, Elem_DOF, Ele_Type, Num_total_layer, Num_global_layer, Num_material, MATRIX S, C;</td>
</tr>
</tbody>
</table>

```c++
GLOBAL_ELEMENT(int num, int type, int EDOF, double H, GENERAL_DATA& GD, MATERIAL& Mat) : ELEMENT(type, num, EDOF, GD), S(6,6), C(6,6):
~GLOBAL_ELEMENT();
void add_info();
void def_Ele_coord();
void definition();
```

Figure 37. Class *GLOBAL_ELEMENT*
Figure 38. Class hierarchy of *GLOBAL_ELEMENT*
One merit of our OOFEA is its capability to accept programs written in other computing languages. For example, in Eq. (3), we have the entities of $L_n$ and $\bar{L}_n$ that contain lengthy algebraic presentation. They are coded in a FORTRAN subroutine. Since the subroutine involves mostly lengthy assignments, it is not necessary to rewrite it in C++. We declare the linkage of the FORTRAN subroutine to OOFEA in the class `GLOBAL_ELEMENT` definition as follows:

```c
extern "C" {
    extern void lxxn_(double* a18n, double* a18bn,
                 double* a34n, double* a3bn, double* a56n,
                 double* shape, double* card, int* mdim,
                 int* nnode, double* hglobal);
}
```

The underscore "_" in `void lxxn_(double* a18n, ...)` reminds the compiler that this is not a C++ function. When we call the subroutine from the C++ code, the underscore is also attached.

4.6 Closure

In this chapter, we presented an object-oriented model for finite element analysis of complex engineering systems. An object-oriented enhanced entity-relationship model has been created for effective processing of a myriad of data types encountered in finite element analysis of complex engineering problems. A class library was presented for object-oriented finite element analysis in general and was applied to a particular problem, interlaminar stress analysis of composite laminates. This research lays the foundation for development of a new generation of highly modular, reusable, and easily maintainable finite element software systems.
CHAPTER V

AN OBJECT-ORIENTED DATA MANAGEMENT MODEL

5.1 Introduction

Data structures encountered in engineering applications, such as finite element analysis (FEA) and computer-aided design/manufacturing (CAD/CAM), are far more complicated than those encountered in business applications. New data models need to be developed for diverse types of data created and used in the engineering world.

We are investigating the application of object-oriented programming in computational mechanics and computer-aided engineering (CAE). As part of this research project, we have developed an object-oriented data management model for managing the input, intermediate, and output data in the computation process. Since the object contains both data and operations within its scope, the data management model is integrated into the analysis procedure of engineering problems. This is beneficial to both numerical computation and data management. On the one hand, repetitious definition of data and data structure is avoided in managing data. On the other hand, computational algorithms are combined with data more tightly and naturally.

The object-oriented data management model presented in this Chapter has been implemented in C++ programming language [Stroustrup, 1991] AT&T Release 2.1 on the Sun SPARC workstation.

In the following sections, we first introduce the object-oriented data management model from the extension of the entity-relationship (ER) data model. Next, we present
details of the data structure of our model. This model is developed for the numerically intensive algorithms frequently encountered in engineering disciplines. Finally, we apply the new model to a commonly used engineering analysis technique, the finite element analysis (FEA).

5.2 Data and Data Models in Computer-Aided Engineering

The data in computer-aided engineering has three notable characteristics: complex internal relationship, variable life-time, and large quantity. Applications in various engineering disciplines, such as civil, mechanical, and electrical engineering, involve a complex internal data relationship. For example, in the process of designing a building, designers must deal with data from different requirements, such as architectural, structural, mechanical, and electrical engineering.

CAE data can be classified as permanent, intermediate, or temporary. Permanent data are those remaining unchanged during the design process. The data from design standards belong to this category. The intermediate data are those necessary from one design phase to the next one. For example, dimensions of a superstructure are part of input data for designing its foundation. The temporary data are those residing in one computation process within a limited time period, and are independent of other processes. As an example, the stiffness matrix of problem domain in finite element analysis belongs to this category. This matrix is no longer needed after the nodal displacements are obtained. In numerical analysis of engineering problems, we often find more temporary data than other two types of data. They are needed only for a limited time during the program execution, ranging from several milliseconds to hours of computing time.

The primary memory of computers (specially microcomputers and workstations) often can not support the large quantity of CAE data while a complicated program is
being executed. Storing the data on the secondary storage devices, such as disks, tapes, and CD-ROMs becomes the only alternative.

Due to the above characteristics, we need to create special data models to deal with complex data involved in CAE applications. A conventional entity-relationship model is not satisfactory for such applications. A few data models have been proposed for this purpose.

5.2.1 Extended Entity-Relationship Model

The entity-relationship model [Chen, 1976] has the advantage of simplicity in representing the real world. The customary table form is the primary tool to represent the real world in this model. For example, in the finite element analysis, the problem domain is divided into sub-domains, called elements (Figure 39). Each element is defined by vertices, called nodes. After inputting nodal coordinates x, y, and z, and other information, the corresponding displacements u, v, and w can be obtained through a series of computations. The data attributes related to the node, that is x, y, z, u, v, and w, are tabulated in Figure 39 (b). This table can be described by a relation schema \( R \) (NODE, x, y, z, u, v, w). The title row of the table is called a relation with attributes NODE, x, y, z, u, v, and w. Data rows are called instances of the relation or tuples. The instances of this relation schema are information about nodes. Instance data of each attribute in the table compose a column.

We view the table in Figure 39(b) from two perspectives: the aggregation and inheritance relationships. We describe the aggregation relationship first, and the inheritance relationship next. A node is composed of three concepts, NODE, coordinate XYZ and displacement UVW (Figure 40). These concepts become entities in the node relation \( R \) (NODE, XYZ, UVW). The XYZ and UVW are not atomic entities. They are in turn relations with three basic attributes (x, y, z) and (u, v, w), respectively. In other
words, the relations XYZ and UVW play the aggregation role in the relation R's definition.

Three coordinates x, y and z determine the position of a node in the space. The inclusion of the displacement information in the node is an extension of the geometrical node used for graphical representation. The relations R and GR (for geometrical relation) have an inheritance relationship, that is R (NODE, XYZ, UVW) inherits attributes of GR (NODE, XYZ) (Figure 40(b)).

The data shown in Figure 39(b) are in simple table form. However, as the same data presented in Figure 40, they become complex data with inheritance and aggregation relationships. The reasons for doing so are: a) simplifying data representation by showing the branches without the details of each branch; and b) reusing pre-defined codes about relations XYZ and UVW in order to save development effort for relation R.

Figure 39. An element in finite element analysis (a) and its data table (b).
Figure 40. The aggregation (a) and inheritance (b) relationships
The conventional entity-relationship model assumes all data are simple without allowing for complex aggregation and inheritance relationships. The extended entity-relationship model (EER) allows inheritance and aggregation relationships among data attributes [Teorey et al., 1986]. We use the EER model to represent complex data in engineering applications. The relation R in Figures 43(a) and 43(b) is in fact an extended entity relational data model.

5.2.2 Object-Oriented Data Model

The object-oriented data model shows potential for modeling and processing CAE data efficiently. Although a standard definition for object-oriented data model has yet to evolve in the database research community, Dittrich [1991] points out three different approaches, the structural, the operational, and the behavioral. In the structural approach, only data are modeled as objects using OOP techniques. In the operational approach, only operations are modeled as objects. In the behavioral approach, both data and operations on data are modeled as objects using OOP techniques.

Many people consider the EER model totally different from the entity-relationship model. The argument is that in the ER model, only simple attributes are presented in table form; in contrast, the EER model covers complex attributes. Some of these attributes have inheritance and aggregation hierarchies involved.

However, such distinction can be challenged in object-oriented programming paradigm. An attribute is nothing but a data. An integer, an integer array, or a user-defined class are all data types. The only difference is that a class is more complicated than an integer from the data structure point of view. Unless it is necessary, they are treated equally as a data type in OOP.

The same explanation applies to the data model. We look at object-oriented data model as an extension of the relational data model. Some attributes in OO data model are
complex. They are the instances of other relations. The rest attributes remain their simplicity. To those simple attributes, the operation of the conventional relational data model are still applicable. Only when the needs to deal with these complex attributes arise, the special operation for object-oriented data model comes in. Therefore, we continue to use many terminologies for the relational data model in the OO data model, except some attributes are complex objects.

We can view the node in the EER model as an object from the perspective of object-oriented programming paradigm. Node is a complex data object with both inheritance and aggregation relationships. It inherits from geometrical node object, and adds displacement object UVW as its component.

A few papers have appeared on the application of object-oriented data model for CAE recently. Spooner et al. [1986] use object-oriented programming concepts to manage three-dimensional geometrical data for computer-aided design of mechanical parts. Powell and Abdalla [1989] propose an interaction technique between application programs and a central database for integrated computer-aided design based on object-oriented programming and data management concepts.

5.3 A New Object-Oriented Data Management Model

We present an object-oriented data management model for CAE on the basis of the extended entity-relationship model. This model is based on the behavioral approach. In the following sections, we first propose a data storage structure for objects containing dynamically allocated data. Then, we introduce a 3+ index system for effective management of data on the secondary storage devices in connection with the aforementioned data storage structure. Finally, we present the data operations of the new model.
5.3.1 Data Storage Structure for Dynamically Allocated Data

In the data storage structure provided by C++ programming language, all data of the superclass are inherited to the subclass. Regardless of being used in subclass or not, the total memory of the subclass includes all the data in its superclass. This results in many unused blank spaces within one object block in the database. If there are several levels of inheritance involved, the subclass at the lowest level carries a large number of data. The drawbacks are: a) an increase in the memory requirement from secondary storage equipment; and b) a decrease in searching speed on the secondary storage. The advantage is that all related data are clustered together. Both updating and retrieving are easy, especially when they are implemented by the read and write methods of fstream. If the address of data on the file is recorded properly, the drawback b) can be overcome significantly.

a) Static Components of An Object

The central feature of an object-oriented programming language, such as C++, is the user-defined data type, class. Since everything in the object-oriented programming paradigm is represented in the form of a class, a data management model in such an environment has to be related to the class structure.

When the object of a class resides in the computer main processing memory, the compiler ensures the encapsulation of data and operations. Even the dynamically allocated data are combined with operations in the objects. However, when we deal with the secondary storage, such as files on the disk, the situation is different. In particular, the dynamically allocated data memory is released when the objects are destroyed.

An object in C++ programming environment can be viewed as one unit during the input and output (I/O) operations. For example, consider a class A as follows:
class A{
    int i;
    double d;
    char c[2];
    ....  // other operations
};

Assuming an object of the class A is \textit{an\_object}. It can be inputted and outputted to the storage device as follows:

\begin{verbatim}
fstream f;
f.open(file_name, file_mode);
f.seekp(ptr_position, count_from);
f.write((char*)&an\_object, sizeof(an\_object));
f.seekg(ptr_position, count_from);
f.read((char*)&an\_object, sizeof(an\_object));
f.close();
\end{verbatim}

where \texttt{f} is an instance of \texttt{fstream} (an I/O library); \texttt{open}, \texttt{seekp}, \texttt{write}, \texttt{seekg}, \texttt{read}, and \texttt{close} are methods of \texttt{fstream}; \texttt{file\_mode} indicates the file is opened for input, output, or append; and \texttt{ptr\_position} is the position of I/O device head on the file from the position \texttt{count\_from}. This code involves several operations: 1) open the file named \texttt{file\_name} in one of the file modes (input, output or append); 2) find the position on the file for I/O device head to write; 3) write the entire object byte by byte on the file [its size is measured by function \texttt{sizeof()}]; 4) look for the position on the file for I/O device head to read; 5) read the entire object byte by byte from the file; and 6) close the file.

An object of Class A is stored in the file as shown in Figure 41 in the sequence of data definition. BOF and EOF stand for beginning of file and end of file, respectively. Starting from the position where the object is stored, integer number $i$ takes 4 bytes, double precision number $d$ takes 8 bytes, and character string $c$ takes 3 bytes (two for
characters and one for character pointer address). In total, this object takes 15 bytes memory space in the file.

The C++ programming language also provides I/O functions for a complex object with inheritance and aggregation relationships. For example, consider class B which is a subclass of class A as follows:

```cpp
class B: public Class A{
    C c_obj;
    ...... // other operations
};
```

Class B inherits data i, d and c from class A, and has its own data c_obj which is an instance of class C with size of 6 bytes. The complexity of class B is due to the inheritance relationship with Class A and aggregation relationship with Class C. The data storage structure of the object instantiated from class B is shown in Figure 41. The data are stored in the order from superclass to subclass. Instances of a class, such as c_obj, are treated the same way as standard data types. Therefore, the data in the file are stored as i, d, c and c_obj sequentially.
Figure 41. Data storage structure of an object of Class A in the file.

Figure 42. Data storage structure for a complex object of Class B in the file.
b) Dynamic Components of An Object

The data storage structure described above works well for the objects of object-oriented data model with static data variables. However, this is not always the case. Dynamic data variables are necessary in CAE applications. The memory space for dynamic data variable is assigned during execution of the program and released when the program runs out of the scope in which the data belongs to. The solution to such a problem is not provided by standard programming language functions. It is one of the issues to be dealt with in developing an object-oriented data management model.

In engineering analysis, we encounter two types of dynamically allocated memory. The first type is the dynamic object array. For example, in the definition of the aforementioned class A, instead of defining c as a string of two characters statically, we can define c as a pointer to the character dynamically, that is char* c. The dimension of this character string will be determined when the object is instantiated. The object of such a class records only the pointer (the start position of the string on the computer memory) of the string as a short integer number (it occupies 2 bytes of memory space compared with 4 bytes for the regular integer number in UNIX environment). When the program is terminated, the memory space that pointer occupies is released and assigned to other objects. Retrieval of such a string becomes impossible.

The second type of dynamically allocated memory is a recursive dynamic object, that is when a dynamic object has another dynamic object as its component. For example, the vector (an array of double precision numbers) and matrix (a two dimensional array of double precision numbers) operations are frequently performed in numerical analysis. VECTOR and MATRIX classes are two of the basic classes required for engineering applications. Their dimensions are not defined statically in the definition. On the other hand, from the mathematical point of view, matrix is an array of vectors, or an array of double precision number arrays. Thus, we define a matrix dynamically as an
array of VECTOR objects, and vector object itself is an array of double precision numbers. Such dynamic allocation of the memory can occur recursively.

We present our solution to the dynamic allocation problem through vector and matrix examples. First, we create a VECTOR class with nine data variables in its definition.

```cpp
class VECTOR{
    ...... // other operations
private:
    Object_type Type = OVECTOR; //vector class
    Bool ACTIVE; //flag
    streampos Offset; //array start position
    streampos Length; //array length
    char File_name[15]; //file name
    int Dim; //dim. of the array
    int form; //output form
    double *Array; //pointer to double
    char Name[15]; //vector name
};
```

Each data is explained in the class definition. The key attribute of this relation is the character string Name which serves as vector name. The first five data (Type, ACTIVE, Offset, Length and File_name) are specially designed for file I/O in our object-oriented data management model. File_name is optional. It can be provided during the database operation. These file-related data are declared at the beginning of data block so that whenever the object is recorded on the file, they are at the top of the block. The data Object_type is a user-defined enumeration data type (a set of constant numbers) which is associated with all classes. For example, we can define Object-type as follows:
typedef enum Object_type{
    NOTHING, OVECTOR, OMATRIX
}:

where enum stands for enumeration, and OXXXX is used for class XXXX to avoid confusion with class name.

An object of VECTOR on a file is composed of two parts, the header and the dynamic component (Figure 43). The header is stored via standard C++ method write((char*) &vector_obj, sizeof (VECTOR)). It is a fixed-length (58 bytes) record. The dynamic component is stored right after the character string Name. This position is recorded as an Offset (a position on the file where a new record starts) which indicates the start position of the dynamic component. The length of the dynamic component is computed and recorded as Length in the unit of bytes. This length changes, depending on the array dimensions. The total length of the object is the summation of the header length and dynamic component length. Each entity of the dynamic array is listed continuously in the file to the end of the object block.

The retrieving operation of the above object with dynamic components is performed similar to that for storing operation. The header part is read by the standard method read((char*) &vector_obj, sizeof(VECTOR)). After the Dim and Offset information about the array are obtained from the header, the array entities can be retrieved one by one from the database.

For the recursive dynamic objects, the above operations are performed recursively. In this case, the dynamic component is another object with a dynamic component itself. We present the class MATRIX as an example of recursive dynamic objects. The definition of class MATRIX is as follows:
Figure 43. Data storage structure for a VECTOR object
class MATRIX{

...... //other operations;
private:
    Object_type Type; //matrix
    Bool ACTIVE; //flag
    streampos Offset; //array start position
    streampos Length; //array length
    char File_name[15]; //file name with 15 char.
    int Rdim; //number of rows
    int Cdim; //number of columns
    int form; //output form
    VECTOR *Row; //pointer to VECTOR
    char NameflSJ[15]; //matrix name
};

Here we define the matrix as a dynamic array of VECTOR objects. The data storage structure for object of class MATRIX is shown in Figure 44. In addition to an extra static data (Cdim, which stands for number of columns), object of MATRIX has a complex dynamic component. The complexity comes from the aggregation of a set of dynamic vector objects. Although objects of class VECTOR are stored one by one continuously, the header of the matrix object only records the position of the first vector object. All other vector objects are assumed to be recorded continuously. The offsets of dynamic array in vector objects are recorded in their own header part, that is at the local level of vector objects.

Figure 45 shows a more general case of recursive dynamic allocation of objects. All objects have dynamic components. There are seven objects in total. Object 1 contains Object 2 and Object 3. Object 2 has Object 4 and Object 5 as its components. Object 3 has Object 6 as its component. Object 6 has Object 7 as its component. Each
branch is clustered together with its root. Objects 4 and 5 are stored as parts of Object 2, Object 7 is stored as part of Object 6, Object 6 is stored as part of Object 3, and Objects 2 and 3 are stored as parts of Object 1.

Figure 44. Data storage structure for a MATRIX object
Notations:

H: Header  L: Length  Offset of dynamic part

Figure 45. Data storage structure (a) for a complex object tree (b)
There are four advantages in such an arrangement. The first advantage is a decrease in the responsibility of each individual object, even when it is a complex object with several levels of hierarchy inside the dynamic component. The second advantage is to free the limitation on number of objects and depth of object hierarchy in the dynamic component. Object at the upper level controls only the first dynamic component, which can be either an array of objects, or an object with another object as its dynamic component. The third advantage is a simplification in the data management operations. For example, during the search process of retrieving a selected node, retrieving the dynamic part (coordinates of the node) is not necessary until the node number matches. Retrieving the header part only can reduce considerable amount of CPU time when the dynamic part is long and complicated. The fourth advantage is the convenience in both depth-first search and breadth-first search in a knowledge base built on the proposed storage structure. This multi-level offset recording system is the core of the index system in our object-oriented data management model, to be discussed in the next section.

5.3.2 3+ Index System

The data storage structure presented in this Chapter for dynamically allocated data is an effective method of data management. However, we need a mechanism for locating the start position of an object on the secondary storage. This is handled by a file manager residing in the main memory.

The file manager of our object-oriented data management model is a conventional relational database with four attributes, \textit{Type}, \textit{Obj}, \textit{Act}, and \textit{Ptr} (Figure 46). The attribute \textit{Type} corresponds to the class type. It is the class identification number. In cases when we know the objects on the file and want to track what class type they instantiated from, this ID number can be useful. The attribute \textit{Obj} shows the identification number of the objects from the same class. In order to reduce the processing time needed to erase the
whole record from the secondary storage, we use the flag \textit{Act} to indicate whether the object record is active or inactive. Even when it is inactive, we can still find the length of memory space it occupies and replace it with an object of equal or shorter length. The final attribute \textit{Ptr} of the file manager indicates the start position of an object on the secondary storage.

The file manager, the object, and the dynamic component collectively make an efficient 3+ index system for the objects on the secondary storage (Figure 47). If there are many objects in the application, it is better to organize the files in accordance with class types. Otherwise, we can include all the objects in the same file, regardless of their class types.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure46.png}
\caption{File manager}
\end{figure}
5.3.3 Basic Operations of Object-Oriented Data Model

The basic operations in the object-oriented data management model are insertion, updating, retrieval, and deletion.

a) Insertion

Insertion operation is performed frequently on the database. Both storing a new object or updating an old object are equivalent to insertion. A necessary criterion for insertion is the protection of database records. This requires checking the available storage space on the secondary storage before insertion.

We developed a simple scheme for checking the requested start position to see whether it is suitable for insertion operation. First, we check whether the user-specified start position is at the end of the file. If it is true, we can insert the object immediately at the end of the file. If not, we have to check whether or not the user-specified slot in the
file has enough memory for storing the entire object. If the length is not enough for the object, we have to move the start position to the end of the file.

b) Updating

Updating is usually done on part of an object's data. If the data is part of the object header, we can compute the position of the data from the object's start position and overwrite the old value with a new value. If the data is part of the dynamic component, we first move to the offset, and then find its position according to its order in the sequence and overwrite it. If the dynamic component is an object, we read the header of this object and find the position of the data to be updated. Updating an entire object is the same as inserting a new object after the old object is deleted by switching the ACTIVE flag to NO.

c) Retrieval

Retrieval of an entire object or selected attributes of the object is the opposite operation of insertion and updating, respectively.

d) Deletion

Deletion of an entire object is achieved by declaring ACTIVE flag in the header as NO. Thereafter, the whole object is viewed as deleted. Deletion of selected data is performed by declaring the data on the header part as NULL. It is impossible to remove the space assigned for the data physically. Because all the data within class definition have their positions on the file. In certain circumstances, NULL might be confused with zero. The user can specify another number instead of NULL to avoid confusion, as long as it is equivalent to the deletion.

5.4 Application in Finite Element Analysis

Finite element analysis is a general method for numerical solution of engineering problems that can be described by differential equations. A complicated problem domain
is divided into simple elements. Complex engineering problems are modeled using hundreds or thousands of finite elements. The huge amount of data generated during the solution has to be managed efficiently. The object-oriented data management model presented in this Chapter is particularly effective for such a problem. To demonstrate the application of this model in finite element analysis, we present the node data management in this section in detail.

As we discussed earlier, nodes play an important role in finite element analysis. First, they determine the shape and position of the elements geometrically. Second, they describe the deformation of the elements after numerical analysis through their displacement attributes. A node database should include two sets of data, the nodal coordinates and the nodal displacements.

Both nodal coordinates and nodal displacements are defined as an array of double precision numbers. They are objects of the class VECTOR. The dimension of nodal coordinates is normally equal to three. However, the dimension of displacement vector is problem-dependent. In some cases, the rotational degrees of freedom have to be included. Therefore, the dimension information is included in the node class definition.

Object identifiers are the key attributes for operations. Two identifiers are used for the node class. They are integer number \textit{NodeNum} and character string \textit{Name}. \textit{NodeNum} is used for numerical computations, especially for array operations. \textit{Name} is used for post-processing after computations.

Another set of data is the storage information on the secondary storage device. They are \textit{Type}, \textit{ACTIVE}, \textit{Offset}, \textit{Length\_xyz}, \textit{Length\_disp}, and \textit{File\_name}. The definition of \textit{ACTIVE}, \textit{Offset}, and \textit{File\_name} is the same as defined before. \textit{Length\_xyz} and \textit{Length\_disp} are lengths of the coordinate object and the displacement object in bytes. The definition of length here is different from that in the class VECTOR. Both \textit{Length\_xyz} and \textit{Length\_disp} include the array header length (size of VECTOR) and array
entity length (double precision number size multiplied by dimension). On the global level, the class NODE needs to know the total length of its dynamic components $xyz$ and $disp$. On the local level, the local objects ($xyz$ and $disp$) know their header size and need to keep only the lengths of their own dynamic components.

The reason for having only one offset instead of two for both coordinate and displacement objects is simplicity. As a rule, all data are recorded on the secondary storage in the sequence they are declared in the class definition. $Offset$ always indicates the start position of the first dynamic component and the end of the header part written by the $write$ method in C++.

Number and types of operations on database are determined by the problem at hand. For example, node deletion operation is not needed in the finite element analysis, though it might be necessary in an integrated CAD/CAE system. After the computation is completed, the whole database setup for the analysis can be removed, including information about nodes, elements, etc. On the other hand, some operations, such as updating, are used frequently.

After the object of a node has been instantiated and essential data are setup, the node object can be inserted into the database. This usually is done during the pre-processing. At this time, however, the displacement information is still not available. It will be inputted as updating after the displacement computation. When node object is recorded, memory space should be reserved for displacements.

The insertion operation is implemented by two methods, $Store$ and $Insert$. The former stores the node object at the exact position assigned. The latter performs the checking procedure described in a previous section. The final start position assigned to the $Store$ method is determined by the $Insert$ method according to the memory space availability. After the operation is completed, it returns the final start position to the file manager for the purpose of book-keeping.
Only displacements need to be updated in the node database after they are obtained from numerical computations. Coordinates of node will not be changed during the computation process. Several parameters are needed for updating the displacements, such as file name, start position of the node object, node name, node number, vector of new values, and/or array of new values. But not all of them have to be used for updating displacements. The index system described previously can track the internal relationships between these parameters to avoid listing all possible parameters for update methods.

A complete class NODE description is listed in Figure 48. Many other methods are needed in the finite element analysis. Some of them are presented in [Yu and Adeli, 1993] and Chapter IV.

5.5 Closure

This chapter presented an object-oriented data management model for engineering numerical analysis. The data storage structure and 3+ index system are the contributions of this model. As part of an on-going research project, the database management system is integrated with the engineering numerical analysis in an object-oriented programming environment. This integration is particularly desirable in computer-aided engineering and manufacturing with an efficient database management technique for managing large quantities of permanent, intermediate, and temporary data.
### Class Name
| NODE |

### Super Class
| VECTOR |

### Composed Class
| VECTOR |

### Data
- VECTOR xyz, disp;
- int NodeNum, NumDim, NumDOF, Status, Disp_data;
- Object_type Type;
- char Name[15], File_name[15];
- Bool ACTIVE;
- streampos Offset, Length_xyz, Length_disp;

### Method
- **private:**
  - void SetUp(int, int, int);
  - NODE& Copy(const NODE&);

- **public:**
  - NODE(void);
  - NODE(int, int, int);
  - NODE(int, int, int*, double ...);
  - NODE(int, int, double*, int ...);
  - NODE(const NODE&);
  - NODE(const NODE&, const NODE&);
  - NODE(const NODE&, const NODE&, const int);
  - ~NODE(void);

  - int get_NodeNum() const;
  - int get_NumDOF() const;
  - int get_NumDim() const;
  - int get_Status() const;
  - int get_Disp_data() const;
  - Bool get_ACTIVE();
  - char* get_Name();
  - char* get_File_name();
  - VECTOR get_xyz();
  - VECTOR get_disp();
  - streampos get_Offset();
  - streampos get_Length_xyz();
  - streampos get_Length_disp();

---

Figure 48. Class card of class NODE with dynamic data management model
### Class Name: NODE (continued)

<table>
<thead>
<tr>
<th>Super Class</th>
<th>Composed Class</th>
<th>Data</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE</td>
<td></td>
<td></td>
<td>void assign_xyz(double *);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>void assign_disp(double *);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>void assign_xyz(VECTOR&amp;);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>void assign_disp(VECTOR);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>void assign_status(int);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>void assign_node_num(int);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>void assign_Num_Dim(int);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>void assign_Disp_data(int)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>friend ostream&amp; operator &lt;&lt; (ostream&amp;, const NODE&amp;);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>NODE&amp; operator +=(const NODE&amp;);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>NODE&amp; operator -= (const NODE&amp;);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>streampos Insert(char*, streampos);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bool Store(char*, streampos);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bool Update_disp(char*, streampos, VECTOR&amp;);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bool Update_disp(char*, streampos, double*);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bool Update_disp(char*, char*, VECTOR&amp;);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bool Update_disp(char*, int, VECTOR&amp;);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bool Retrieve_disp(char*, streampos);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bool Retrieve_disp(char*, char*);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bool Retrieve_disp(char*, int);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bool Retrieve_xyz(char*, streampos);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bool Retrieve_xyz(char*, char*);</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bool Retrieve_xyz(char*, int);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Non-member Function</th>
<th>NODE operator + (const NODE&amp;, const NODE&amp;);</th>
</tr>
</thead>
</table>
CHAPTER VI

AN OBJECT-ORIENTED VERSION MANAGEMENT MODEL

6.1 Introduction

The complexity of both data and operations involved in computer-aided engineering (CAE), such as building or machine design, or product manufacturing, is one of the driving forces in the development of object-oriented database management systems (OODBMS), in an attempt to deal with complex data, their relationships, and operations on them efficiently.

During the past few years, OODBMS has become one of the most active research areas in the computer science community. OODBMS researchers, however, have mostly concentrated on business applications, and only recently have moved to other areas, such as design of very large scale integrated (VLSI) circuits [Katz and Chang, 1991]. Little attention has been given to computer-aided engineering in disciplines such as aerospace engineering, civil engineering, and mechanical engineering. The CAE problems in these engineering disciplines have their own characteristics and need special data management techniques to meet their requirements.

One characteristic of computer-aided engineering is the extensive numerical analysis involved in almost all steps of design and manufacturing. However, OODBMS algorithms or models developed so far, for example for design of VLSI circuits, are not appropriate for other numerically intensive CAE applications. Research has been reported on the version management problem related to time history [Ahmed et al., 1991]
and class inheritance [Katz and Chang, 1991]. The version management of design objects with alternative data has not been studied.

This Chapter presents a new version management model specially suitable for CAE applications involving extensive numerical analysis. Along with the introduction of the basic structure of this version management model, we present implementations of the model as well. Finally, we illustrate the application of this version management model in conjunction with dynamic object manager in the complex problem of composite laminate analysis.

6.2 Version Management in Computer-Aided Engineering

Before discussing object versions, we first define several terms used in this Chapter. A design class is a class defined broadly for representing one subject and operations on this subject in computer-aided engineering. For example, a stiffness matrix formulation is a subject in finite element analysis, and a beam design is a subject in structural design. The object instantiated from the design class is called design object. Data includes simple data type and user-defined data type (including class). Simple data is also called atomic data, such as integer and character. User-defined data types are non-standard data types defined by users, such as matrix class and vector class. In the definition of a design class, data means objects of data type which includes both simple data and user-defined data. Different data objects can be used for instantiating the same design class. These data objects are called alternative data objects. Objects instantiated by user-defined data objects are complex objects often with inheritance and aggregation relationships.

Version means different things in various disciplines. In software development, version is used for recording the class evolution over time history. The time stamp is a primary identifier to the version and is generated by the computer system clock. In
business oriented database systems, version is related to the transaction of a database at
certain time. Time stamp reflects the time history of the data record. In engineering
design process, version refers to an alternative to the design object. One group of data
makes one version of design objects.

Engineering design is a trial-and-error process. New objects are instantiated from
the same design class continuously to meet the various design requirements and
constraints. Each object is one version of a design object set. For example, in designing
a steel beam, all the necessary design parameters are defined in the design class of the
software system. A design engineer inputs different sets of data to test whether the beam
satisfies all the design requirements, and is compatible with surrounding columns, joints,
and floor systems designed separately. Under certain circumstances, a design engineer
may want to review the design object several revisions back from the current version.
The best version of design objects is selected by the engineer as the final product. These
procedures require a record-keeping mechanism in terms of version management in the
design process.

Another application of version management is in integrated design systems, for
example a system that integrates architectural design, structural design, mechanical
design (HVAC), and construction management for design and construction of buildings.
Each phase retrieves one version of design data from the central database [Powell and
Bhateja, 1988], and stores it in its private database for further processing. When the
design is completed, the revised data are put back to the central database as a new
version.

Considering that there are usually a number of unchanged parameters in design
objects during the entire design process, some researchers view other versions (objects)
as revisions of the previous design version (object). Each revision is represented by
inheritance from the old version [Kim, 1990] [Katz and Chang, 1991]. However, since
the current object-oriented programming languages do not support such object-to-object inheritance, the implementation techniques are still under investigation.

6.3 Issues in Version Management Models

Version is one of the key topics in object-oriented database management systems. Researchers have proposed many version management models and operations in the object-oriented database management systems, for example Katz et al. [1986], Chou and Kim [1986], Dittrich and Lorie [1988], and Ahmed and Navathe [1991]. In this section, we will discuss several common issues in these version management models.

6.3.1 Object Version Tree

Almost all version models employ an object version tree (OVT). Setting aside the differences in the description of details (for example, class derivation version, object alternative version, instance-to-instance inheritance version, etc.), an object version tree starts from a design object and grows in two directions: path and derivative (Figure 49). Design object in this figure has three paths, V1, V2, and V3. The term V and its associated digital number stand for version and version stamp (version identifier), respectively. Version V2 has two paths V2.1 and V2.2. They are the derivatives of V2 and inherit some values from V2. Version V2.1 further has its own two paths V2.1.1 and V2.1.2. The inheritance relationship between parent version object and child version object is kept in the version stamp. For example, in V2.1.2, the first 2 (two) indicates the parent version V2, 1 (one) indicates path 1 of V2, and the second 2 (two) indicates path 2 of V2.1. As the number of paths increases, this version tree can grow wider and deeper.
Figure 49. Object Version Tree

Legend:

- Design class
- Alternative data class
- Design object version
- Alternative data object

Figure 50. Paths to Versions
The relationship between a design class and its design object versions is shown more clearly in Figure 50 [Dittrich and Lorie, 1988]. In this figure, the design class has two alternative data (un-fixed data) classes as its components, and each alternative data class has two alternative data objects. Thin lines starting from the design class and ending at the design object versions are called paths. With different alternative data objects, different design objects are created from the design class. These design objects are called (object) versions of the design class. In many object-oriented database system presentations, the instantiation of design objects from the design class is shown with thick lines, and paths through alternative data are omitted for the sake of simplicity.

Figure 50 shows a simple version problem concerning two alternative data types only. However, data types are not limited to pure data. In some cases, the data type itself is a design class. For example, in a framed structure, beam and frame are two design classes. But, frame can take beam as its data type. Thus, different structural frame objects are created by various beam objects. In such cases, the design class is simply viewed as an alternative data class.

6.3.2 Time Stamp

Some versions evolve in time history. Time of creating a version object is a useful index for keeping track of version evolution. This index is called time stamp. or simply stamp if not confused with the version stamp used in the object version tree (Figure 49). For example, we can re-name versions of the design object in Figure 49 with their creation time (shown in parentheses), the elapsed time (in seconds), based on the computer system clock. Time stamps help to distinguish the current version from previous versions for time sensitive version management. Obviously, the time stamp does not convey derivation information of various versions. The object version tree can not be created from the time stamps alone.
6.3.3 Versions at Class Level and Object Level

Two types of version can be recognized, one is at the class (type) level and the other one is at the object (instance) level [Ahmed and Navathe, 1991]. Although most version management models proposed so far focus primarily on the object level version, class level version should become increasingly attractive. Because software vendors are developing more general purpose database management systems with undefined class structures (for both data and operations), so that users can have more flexibility in modifying the software for their specific usage.

In general, version at the class level means different internal structures in class definition. This is of concern in the development and maintenance of OODBMS. This kind of version management model should be part of a high level computer-aided software engineering (CASE) tool. Version at the object level means different values to the same data attributes of the design objects instantiated from the same design class. This is what OODBMS users deal with when using application programs. The mechanism of version at the object level should be provided by the database management system. In the rest of this Chapter we focus on the version problem at the object level.

6.4 A New Object-Oriented Version Management Model for Computer-Aided Engineering

In this section, we present a new object version management model for computer-aided engineering. First, we point out the shortcomings of object version tree for objects with alternative data. Then, we observe the nature of object versions in numerically intensive CAE, i.e. trial-and-error with alternative data. Subsequently, we propose a parallel version graph to replace the object version tree. And finally, we present the structure of our new version management model for computer-aided engineering in detail.
6.4.1 Characteristics of Object Version Tree

We consider the object version tree as a good graphical representation of version relationships for class evolution. However, OVT is not effective for the version management at the object level. Our arguments are as follows:

1) The version tree emphasizes the inheritance relationship. This relationship exists between superclass and its subclasses, but not between objects and their classes or other objects.

2) The CAE users of OODBMS currently deal with software systems with pre-defined classes only. They are interested in objects instantiated from the same design class and relationships of these design objects. This is where the version management model can play a useful role in CAE.

3) There is no inheritance relationship among objects instantiated from the same design class. The differences among objects instantiated from the same class are values of some or all re-definable attributes (alternative data as mentioned previously). If some re-definable data attributes carry the same value in two different objects, it does not mean one object inherits from another object. These two objects are equivalent hierarchically. If the same value of one data attribute appears in all objects, it can be defined with the class definition so that all objects can share this value.

4) Design is a trial-and-error process and the most desirable solution is not always the last revision (version). A hierarchy tree gives the wrong impression that design continuously improves as the branch gets deeper.

5) Versions are clustered together according to their data attributes in OVT. Sequence of version creation is unclear unless a time stamp is used. Such a clustered data structure slows down the search procedure.

6) For a multiple data attribute case, the object version tree can become huge and difficult to manage.
7) Repetitious versions may be generated on different paths of OVT, because the same data attribute value could appear in several branches.

6.4.2 Characteristics of Object Versions in CAE

In order to find out the nature of objects and their versions in computer-aided engineering, we will take a closer look at data, classes and, objects in the design process. In computer-aided engineering, the relationships among data, classes and, objects are analogous to that of raw materials, machines, and products. Machines turn raw materials into products. Changes of the raw material result in a different product. Materials and products have one to one relationship.

Version is an index showing the intermediate status of the design process. Let us consider a simple design system shown in Figure 51 as an example. The goal of this system is to come up with an acceptable design object using the data and classes provided. There are three atomic data objects (Data 1, Data 2, and Data 3), three data classes (Class 1, Class 2, and Class 3), five alternative data objects (C1.1, C1.2, C2.1 C3.1, and C3.3) and many design object versions (only five versions V1 through V5 are shown in figure) from one design class. Each version is a design object instantiated from the design class. The design object is composed of one or two alternative data objects. Alternative data objects are named after their data class name with an extension of atomic data number to show their roots. For instance, C2.1 means an alternative data object instantiated from data class Class 2 using atomic data Data 1. Versions V1, V2, and V4 use one alternative data object only. Versions V3 and V5 use two alternative data objects.
Figure 51. Relationships among data, class, object and version
Several characteristics of object versions in computer-aided engineering can be observed in Figure 51. 

1) Design is a process of trial-and-error. Design objects V1 and V2 are instantiated from the same design class using data objects C1.1 and C1.2 which are different objects of the same data type. Based on the design requirements, the design system determines which version of design objects, V1 or V2, is more appropriate.

2) In some design phases, there are more than one data object involved, such as versions V3 and V5. Changing one data object (e.g., C2.1 replacing C1.2) makes two different design object versions.

3) The sequential numbers of objects and versions are enough to distinguish different objects and versions. The new version carries a larger number. Time stamp is not commonly used in CAE. Engineers feel more comfortable to identify both objects and versions through sequential numbers or brief descriptions (for example, a short name in English).

4) Two versions can share the same data object (e.g., data object C3.3 in versions V3 and V5). There is no inheritance relationship between these two versions. They are parallel or equivalent hierarchically.

6.4.3 Parallel Version Graph

In order to represent the true nature of object version in computer-aided engineering, we propose the parallel version graph (PVG) shown in Figure 4. This graph emphasizes the parallel position and the equivalence of design objects instantiated from the same design class with alternative data objects. These design objects are at the same level hierarchically and inherit class properties equally.
The design class in Figure 52 has three data attributes (data objects), one is fixed and two are re-definable. Values of re-definable data attributes are assigned during the initialization of objects. With different combination of fixed and re-definable data attributes, seven different design objects of the same design class are instantiated. They are different versions of the design class.

![Parallel Version graph](image-url)
6.4.4 Object Version Manager

The goal of keeping version information in CAE is to record various options (alternative data) and their corresponding results (design object versions). Basically, changing one or more alternative data can result in one design object version. Each object version should maintain the information about its data objects, especially its identifiers. There are three key parameters in our object version model, version identifier, object identifier (the atomic data is considered an object too), and class identifier (for both design class and data class). An integer number is usually chosen as these identifiers to simplify the computer processing effort.

Our object version manager (OVM) is composed of three main entities: Object_Index, Object_Record and Version_Record (Figure 53). Entity Object_Index keeps record of all objects (both data objects and design objects) participating in the design process, their class types (Class ID), and total number of objects instantiated from the same class (Total Object). This index can help assign a unique global identifier to the object; the newest object is assigned the largest identification number of its class. Since the object identifier is always associated with the class identifier, two objects instantiated from different classes can have the same object identifier.

Entity Object_Record provides class type (Class), object identifier (Obj), object status (Act) and position pointer on the secondary storage device (Ptr). The sequence of entity instances on this record is not important, because a unique object can be retrieved from this record according to its class type and object identifier. Any slot on the record, which is left blank by deletion of another object, can be filled with a new object immediately.

Entity Version_Record has two simple attributes: version number (No.) and version name (Name), and another attribute T (a table surrounded by shadowed boundary in Figure 53) about all objects related to this version. Attribute T has class type (Class)
and object identifier (*Object*) as its sub-attributes. Only those design objects with alternative data need to be recorded on the *Version_Record*. In many engineering problems, the solution algorithm usually assigns local and global identifiers to objects in order to organize the computations. Since it is possible to use one object more than once in one version, the *Version_Record* assigns a local object identifier in addition to the global object identifier used in all three entities of OVM. Attribute *T* is an array with local object identifier as array number and global identifier as array contents from the data structure point of view.

![Figure 53. Data structure of object version manager](image)
6.4.5 Operations

Four basic types of operations are necessary for the Object Version Manager: creation of version, copying of objects from one version to another version, retrieval of an object of a specified version, and inputting/outputting (I/O) of object version manager information on the secondary storage.

Version creation

At each iteration of the design process, one design object is created and a version number is assigned. After an object is instantiated from a class, this object is immediately assigned a global identifier by OVM. Positions of objects on the secondary storage and their class types are recorded on the Object_Record. If this design object has alternative data, the information of the design object and its data objects is recorded on the Version_Record.

Copying objects from one version to another version

In some cases, two versions share some of all data objects. In these cases, the OVM simply copies the T record of old version to the new version in the Version_Record so that two versions point to the same data objects on the secondary storage. No direct byte-by-byte copy operation on data objects is necessary.

Retrieving an object

To retrieve an object, the OVM first checks the Version_Record in order to find the appropriate global object identifier using the class type and local object identifier. Then, this global identifier is used to obtain the position of the object on the secondary storage with the help of Object_Record.
**Inputting and outputting of OVM data**

Some parts of the design process take considerable amount of computer processing time. After each trial-and-error process, it is desirable to archive all object and version information. With the information provided by OVM data, the designer can turn on the system and continue the previous day's work. The OVM information is stored separately with the detailed information about objects for the sake of expandability and convenience of OVM data file.

### 6.5 Implementation of Object Version Manager

This section presents the implementation aspects of our object version management model in C++ programming environment [Stroustrup, 1991]. A number of presentation methods have been proposed in the literature to illustrate object-oriented analysis/design (OOA/OOD) considerations. We use the modified class card [Yu and Adeli, 1991] to record information for implementation. Six components in our modified class cards, Class Name, Super Class, Composed Class (to list those classes having their objects used as members in this class), Data, and Method serve the basic requirements of representing a class. Data and methods of a class usually are secured in three levels to provide accessibility by other methods: public, protected and private. The default data type is private.

Our version management model is implemented in three classes: **Object_Record**, **Version_Record**, and **OVM** (object version manager). The first two classes directly correspond to the two entities **Object_Record** and **Version_Record** described in the previous section (see Figure 53). Four attributes of **Object_Record** become four private data in class **Object_Record** (Figure 54). The last method shown in the card displays the content of **Object_Record** on an output device. All other methods are self-explanatory.
<table>
<thead>
<tr>
<th>Class Name</th>
<th>Object_Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Class</td>
<td></td>
</tr>
<tr>
<td>Composed Class</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>int Type, Obj, Act; streampos Ptr;</td>
</tr>
<tr>
<td>Method</td>
<td></td>
</tr>
<tr>
<td>public:</td>
<td></td>
</tr>
<tr>
<td>Object_Record();</td>
<td></td>
</tr>
<tr>
<td>~Object_Record();</td>
<td></td>
</tr>
<tr>
<td>void assign_Obj(int);</td>
<td></td>
</tr>
<tr>
<td>void assign_Act(int);</td>
<td></td>
</tr>
<tr>
<td>void assign_Type(int);</td>
<td></td>
</tr>
<tr>
<td>void assign_Ptr(streampos);</td>
<td></td>
</tr>
<tr>
<td>int get_Obj();</td>
<td></td>
</tr>
<tr>
<td>int get_Act();</td>
<td></td>
</tr>
<tr>
<td>int get_Type();</td>
<td></td>
</tr>
<tr>
<td>streampos get_Ptr();</td>
<td></td>
</tr>
<tr>
<td>friend ostream&amp; operator&lt;&lt;(ostream&amp;, const Object_Record&amp;);</td>
<td></td>
</tr>
</tbody>
</table>

Figure 54. Class card of class Object_Record

Except for attribute T and its related methods, class Version_Record (Figure 55) is similar to class Object_Record. Attribute T is defined as a two dimensional array. The dimension can be either pre-defined in the class or dynamically defined through object instantiation. We show T[5][10] in the class card for the convenience of presentation only. For each version, there are five different class types as alternative data type, and each class may instantiate up to ten objects. As an example, a new object i1 of class type i2 is added to the Object_Record as local object i3 via method void assign_T(int i2, int i3, int i1), and the total number of objects for class type i2 is increased via method void Add_T(int i2) by one.
<table>
<thead>
<tr>
<th>Class Name</th>
<th>Version_Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Class</td>
<td></td>
</tr>
<tr>
<td>Composed Class</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>int Ver;</td>
</tr>
<tr>
<td></td>
<td>char Name[15]:</td>
</tr>
<tr>
<td></td>
<td>short T[5][10]:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>public:</td>
</tr>
<tr>
<td>Version_Record();</td>
</tr>
<tr>
<td>~Version_Record();</td>
</tr>
<tr>
<td>void assign_Ver(int);</td>
</tr>
<tr>
<td>void assign_Name(char *s);</td>
</tr>
<tr>
<td>void assign_T(int, int, int);</td>
</tr>
<tr>
<td>short* get_T(int);</td>
</tr>
<tr>
<td>int get_Ver();</td>
</tr>
<tr>
<td>int get_T0(int);</td>
</tr>
<tr>
<td>char* get_Name();</td>
</tr>
<tr>
<td>friend ostream&amp; operator&lt;&lt;(ostream&amp;, const Version_Record&amp;);</td>
</tr>
<tr>
<td>protected:</td>
</tr>
<tr>
<td>void Add_T(int);</td>
</tr>
</tbody>
</table>

Figure 55. Class card of class Version_Record

Class OVM (Figure 56) is the core of our object version manager. It contains objects of classes Object_Record and Version_Record in the form of object array, so that a series of objects and versions can be recorded. Entity Object_Index described previously (see Figure 53) is implemented as a two dimensional array. The size of one dimension is fixed as 2 and another dimension is allocated dynamically. The entity short (*Object_Index) [2] is a pointer to an array of 2 short integers. The size of dynamic dimension is determined by the application program to include all possible objects in the design process.
### Class Card of Class OVM

<table>
<thead>
<tr>
<th>Class Name</th>
<th>OVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Class</td>
<td>Object_Record, Version_Record</td>
</tr>
<tr>
<td>Composed Class</td>
<td>Object_Record, Version_Record</td>
</tr>
</tbody>
</table>

#### Data
- `int Num_Ver, Num_Obj;
- `Version_Record *VM;
- `Object_Record *OM;
- `short (*Object_Index)[2]

#### Method
- **public:**
  - `OVM();
  - `-OVM();
  - `void assign_OVM(Object_type, int, streampos);
  - `void assign_Num_Ver(int);
  - `int get_Num_OM() const;
  - `int get_Num_VM() const;
  - `Version_Record& get_VM(int);
  - `Object_Record& get_OM(int);
  - `short get_Object_Rec(Object_type);
  - `void Copy_VM(Object_type, int, int);
  - `streampos get_OM(Object_type, int, int);
  - `Boolean Store();
  - `Boolean Retrieve();
  - `friend ostream& operator<<(ostream&, const VOM&);

- **protected:**
  - `void assign_VM(Object_type, int);
  - `void assign_OM(Object_type, streampos);

---

Figure 56. Class card of class OVM

One version of design is processed at a time. A check is made to find out whether some objects of this version are the same as objects of other versions. If this is the case, such information is recorded through method `void Copy_VM(Object_type, int, int)`. This prevents any data redundancy on the database. If that is not the case, new objects are created and recorded through method `void assign_OVM(Object_type, int, streampos)`. Method `void assign_OM(Object_type, streampos)` records all objects, their class types.
status and physical position on the secondary storage. If a data object is version
dependent, it is further recorded on Version_Record through method void assign_VM
(Object_type, int). Methods Boolean Store() and Boolean Retrieve() perform storing and
retrieving functions of all records to and from the database.

6.6 Application of Parallel Object Version Management Model

In this section, we present an application of the new parallel object version
management model to a complex CAE problem, the finite element analysis of composite
laminates. The composite laminate is usually made of several layers of glass-epoxy like
fiber materials, called laminae (Figure 57). Each lamina is a layer of fiber arranged in
one direction. It has higher strength in the fiber direction, and lower strength in the other
two directions. Two neighboring laminae are compressed together with a layer of matrix.
Laminae of the laminate can be made of different materials and layered in different

Figure 57. A composite laminate under external force F
angles. The interlaminar stress ($\sigma_z$ in Figure 57) under external in-plane forces can cause separation of lamina and matrix, called delamination of the composite laminate. The delamination symbolizes the failure of composite laminates. To predict the normal stress $\sigma_z$ caused by external forces, Pagano and Soni [1983] proposed a global-local model in order to reduce the computations. The model declares part of a laminate as the global region in which no detailed stress analysis is necessary. The remaining part of the laminate is called local region where a detailed stress analysis is performed.

![Diagram](image)

**Legend:**

- Normal Class
- Virtual Class
- A inherits B
- A uses B

Figure 58. Class relationships in formulating global region stiffness matrix
We developed a finite element analysis approach to the global-local modeling of composite laminate analysis [Chaturvedi et al., 1992]. The core of the system is the formulation of the stiffness matrices for solving the interlaminar stresses and strains. Several input data are critical to the solution of the problem, and are chosen as alternative data for the trial-and-error solution process. These alternative data are: general data (problem parameters, such as the number of layers in the global region, number of materials used in the laminate, element types, etc.), nodal coordinates, node sequence of the element, and material property. These alternative data contribute to all versions of problem solution in the trial-and-error process. They are not atomic data, but are defined in terms of class: GENERAL_DATA, NODE, ELEMENT_SEQ, and MATERIAL. The relationships of the global region stiffness matrix class GLOBAL_ELEMENT with these classes are either inheritance or aggregation (Figure 58). Class ELEMENT in the figure is a virtual class (a class only used as a superclass without object instantiation) in our general purpose object-oriented finite element analysis class library.

Figure 59. Element configuration for the global-local model
We consider two versions in this case (Figure 59): Version 1 uses four Q4 (four-node quadrilateral) elements. The materials used in the global and local regions are different. Version 2 uses four T3 (three-node triangular) elements. The materials used in the global and local regions are the same in this version, but different from those in Version 1.

The content of the OVM for this particular example is shown in Figure 60. Four class identifiers 1 to 4 are used to identify classes GENERAL_DATA, NODE, ELEMENT_SEQ, and MATERIAL, respectively. From Total Object of Object_Index we can observe that there are two general data objects, fourteen node objects, eight element objects, and three material objects in association with two versions.

Although nodes in Version 2 are at the same geometric position as some of the nodes in Version 1 (see Figure 59) and hence have the same coordinates, they are still assigned different object identifiers, because our node class definition includes not only coordinates but also stresses and displacements. The finite element analysis solution depends on many factors, including element type, mesh arrangement, global and local region division, material properties and, so on. The nodal displacements and stresses are not the same in these two versions. Therefore, there are fourteen node objects in two versions altogether.

Only portions of objects in this example are listed in Object_Record shown in Figure 60 for the sake of brevity of presentation. The primary purpose of this record is to locate objects on the secondary storage. The letter "Y" means active record. Numbers in the P tr (pointer) column show the offset (position of object record head) on the secondary storage. For example, 16868 shows the position of the ninth object of NODE class on the file (16868 bytes from the beginning of the file).
The `Version_Record` registers all objects involved in one specific version. The upper portion is for Version 1, and the lower portion is for Version 2. The shadow part means no objects are associated with these slots. For example, in the rows marked as Class 4 (MATERIAL), there are only two and one objects in Version 1 and 2, respectively. The global object identifiers are recorded in the `Version_Record`, for example in the rows of Class 1 (GENERAL_DATA), object 1 for Version 1 and object 2 for Version 2. The global object identifiers recorded on the `Version_Record` are associated with objects on
the *Object_Record* so that retrieval and modification on these objects are straightforward. For example, to retrieve the fifth node in Version 2, its global object and class identifiers are found as 13 and 2 (the thirteenth object of class NODE), respectively. With this information, the *Object_Record* is checked to find out whether this object is an active object. Then, its location can be found at 19284 bytes away from the beginning of the secondary storage.

6.7 Closure

This chapter presented a new object version management model for numerical analysis in computer-aided engineering based on a parallel version graph, which truly reflects the characteristics of design objects with alternative data. This version management model is composed of three collaborating entities, *Object_Index*, *Object_Record*, and *Version_Record*. They are called collectively the object version manager (OVM). The model has been applied to finite element analysis of composite laminates and can be integrated into an object-oriented database management system for CAE applications.
CHAPTER VII

A CONCURRENT APPROACH TO
OBJECT-ORIENTED FINITE ELEMENT ANALYSIS

7.1 Introduction

The goal of developing software systems in computer-aided engineering is
twofold: a well developed software in terms of ease of maintenance and modification, and
a short execution time to obtain the solution. For the first requirement, the object-
oriented programming (OOP) shows distinct advantages over the conventional structured
programming [Booch, 1991]. For the second requirement, parallel processing provides
the right solution [Adeli, 1992a and 1992b]. A combination of these two technologies,
namely concurrent object-oriented programming (COOP) [Agha, 1990], is the best
approach to achieve the above goal.

For solving a complex engineering problem, blackboard architecture provides a
systematic approach to divide a large problem into sub-problems. In this chapter, we
present a concurrent object-oriented programming model for computer-aided engineering
using the blackboard architecture. The finite element analysis of composite laminates is
performed by this model as an application after detailed presentation of the model.

7.2 Concurrent Object-Oriented Programming

Object-oriented programming is basically a methodology for developing software
systems more efficiently. By taking advantage of OOP features, such as modularity,
information hiding, polymorphism, inheritance, and data abstraction, an OOP software
system is much easier to debug, maintain, and modify in the software life cycle. Research on computer-aided engineering (CAE) applications of OOP is growing, for example Baugh and Rehak [1989], Forde and Stiemer [1989], and Miller [1991]. Reviews of basic concepts of OOP as applied to CAE systems can be found in Lee and Arora [1991] and Yu and Adeli [1991].

The motivation for developing concurrent programming systems is to speed up the computational processing. Concurrent object-oriented programming is a new technology which is currently being pursued actively by researchers. Since COOP is still in its infancy, the definition of COOP has yet to be agreed upon by researchers. Yonezawa and Tokoro [1987] define COOP as "a collection of concurrently executable objects and the interactions among the system components are represented by message passing".

The first interaction of concurrent programming with OOP was the synchronization algorithm *monitor* proposed by Hoare [1974]. Monitor is a semaphore with information hiding characteristic. A semaphore is an integer variable proposed by Dijkstra [1965] for exchanging synchronization information between processes. The Actor model [Agha, 1986], a concurrent OOP model developed in 1980's, has had a large impact on several newly developed COOP systems, for example Act 1 [Lieberman, 1987] and ES-Kit [Chatterjee et al., 1991].

More than eighty object-oriented or object-based programming languages are reported in the literature [Saunders, 1989], for example Ada, Common Lisp Object System (CLOS), C++, Eiffel, Objective C, Object Pascal, Smalltalk, and so on. Only a few of them have been further developed with concurrent processing capability, such as PC++ [Lee and Gannon, 1991] based on C++, and ConcurrentSmalltalk [Yokote and Tokoro, 1987] based on Smalltalk. But, no commercial COOP language exists at the time of this research. Thus, we simulate a COOP environment in this investigation.
7.3 Blackboard Architecture

The blackboard architecture was first introduced by Newell [1962] and implemented in Hearsay-II project for speech recognition. The latest version, Hearsay-III [Erman et al., 1988], eliminates the time dimension of Hearsay-II and makes the blackboard architecture a domain-independent problem-solving architecture. Since Hearsay-III, its concept has been applied to various knowledge-based expert systems in engineering, for example Aravind et al. [1991].

The original blackboard architecture is composed of three major components (Figure 1 in Chapter II): knowledge sources (KS), a blackboard (BB), and a controller (also called a world model). A knowledge source can be a knowledge-based system, a neural network learning system, a numerical problem solving algorithm, a group of design criteria, or a combination of them. In the case of knowledge-based system, it is further divided into two sub-components: rules (domain knowledge) and an inference engine to fire these rules. The inference engine can be treated as another knowledge source. Thus, the entire knowledge source becomes a knowledge source hierarchy.

Each knowledge source is activated without explicit order. There is no interruption until one KS is completely executed. As the execution of knowledge sources continues, the hypothesis on the blackboard will be verified and the desirable solution will be achieved. This convergence process is illustrated by evolution from ellipses to a circle in Figure 1. Each subsequent KS works on the basis of the former KS. If one KS discovers an error made by the former KS, it corrects the error and builds up a new basis for the next knowledge source. The whole process of activating knowledge sources is a problem-solving cycle. It is analogous to the jigsaw puzzle game [Englemore and Morgan, 1988]. Each KS participates in the process. The result comes from the cooperation of all knowledge sources during the execution process.
The blackboard is a shared database for posting and modifying hypotheses among knowledge sources. Independent knowledge sources use this platform to exchange information and interact with each other. When one knowledge source is activated by the controller, it reads the information from the blackboard, compares the information with its knowledge, and selects an action: it either modifies the existing hypothesis on the blackboard or creates a new hypothesis. As the cycle of activating knowledge sources continues, we get closer to the solution of the problem.

The blackboard architecture has many advantages in software development: a) organizing knowledge into independent knowledge modules; b) representing knowledge in each module differently; c) using a different inference engine in each module; d) being a flexible representation structure to various applications; e) using multi-levels of abstraction; f) allowing co-existence of inconsistent solutions; and g) developing software systems incrementally in a modular fashion [Aravind et al., 1991] [Lander, 1992].

7.4 A Blackboard Architecture for COOP

Although the blackboard architecture was originally proposed for artificial intelligence applications, it can also be considered as a domain-independent methodology for general-purpose CAE software development. Each KS is an independent entity. As long as the data and hypotheses the KS needs are on the blackboard, it can be executed at any time. Therefore, the blackboard architecture is suitable for extracting parallelism in knowledge sources, information transferring, and data partitioning [Nii et al., 1988]. Furthermore, if we consider each KS as a simple part of a complex engineering problem, the blackboard architecture corresponds to the divide-and-conquer paradigm in parallel computing. We propose a concurrent object-oriented programming model for CAE applications using the blackboard architecture. This model extends the blackboard architecture by adding additional components, so that it can perform complex numerical
computations in solving engineering problems. The basic components of the proposed model are as follows: a hierarchy of tasks, controllers, knowledge sources, global and local blackboards, controlling data slots, and numerical data slots, as shown in Figure 61.

Tasks are categorized in levels. One is at the global level that controls lower level tasks. The level indicates the position of a task in the hierarchy. The global level task has a supreme controller and its assistant knowledge sources. The local level task controls other lower level tasks in the hierarchy. There is no limitation on the expansion of sub-local levels when necessary. This multi-level arrangement of tasks leads itself to concurrent processing effectively. Each task is executed on one processor as a process. The task at the global level controls all the tasks at the next lower level. Local level tasks are assigned to one processor as a thread or several processors separately. This is primarily determined by the availability of processors and dependency of data/objects in each task. For example, in the finite element analysis, each element stiffness computation is arranged as a task.
Figure 61. A general blackboard architecture for CAE applications
Figure 61 (continued)

Legend:

- Original hypothesis
- Improved hypothesis
- Perfect hypothesis (solution)
- Blackboard
- Numerical Data Slot
- Knowledge Source
- Controller
- Module Task
- Controlling Data Slot
- Start
- End
- Processor
- Shared Memory
- Connection
- Process (thread)
- Fork
- Pipe
- Queue
- Semaphore
- Database
- Boolean
- Function
- Join

Y

No
Each task consists of a controller and one or more knowledge sources (Figure 61). The function of the controller is to activate knowledge sources in a certain sequence and evaluate the results (hypotheses) on the blackboard. The controller is the headquarters of the entire task. After the task is accomplished, the controller terminates the activity of the task and sends back a signal to the KS in the upper level. In case of unpredictable situations, the controller itself is monitored by the system user through the user interface of the computer. The knowledge source is an open concept in our model. It is not limited to the heuristic knowledge defined in most traditional artificial intelligence systems [Adeli and Balasubramanyam, 1988]. In addition, we consider mathematical equations, design standards, and experimental data as pieces of knowledge in this architecture. This gives us more flexibility in applying blackboard architecture to CAE applications.

There are one global blackboard and a group of local blackboards in our model. To satisfy the trial-and-error nature of CAE/CAD problems, we design a group of blackboards to conduct version data management. The global blackboard is used by the global level task. The data for controlling all local level tasks in each version are written on one of these blackboards.

Each blackboard is divided into smaller sections (Figure 61). Horizontally, it is divided into a number of sections equal to the number of local tasks. The data accessibility in one task from other tasks should be determined during the task design process. Vertically, the blackboard is divided into two sections, the first section is for storing regular numerical data, and the other section is for controlling data and general information. In an application software system, part of the blackboard can be a database, and another part can reside on the computer main memory. There is no strict format for the blackboard in our model.
All tasks in one version share the same blackboard. This makes data management during the computation easier. Data posted on the blackboard should correspond to the task hierarchy. For example, if two tasks have inheritance relationships, the ancestor task's data are inherited to descendent task. If data in these two tasks are included on two different blackboards, there will be an accessing problem, and additional information about this inheritance relationship should be stored. To avoid this, we use one blackboard in each version.

Applying OOP concepts in managing data on the blackboard is desirable, because OOP defines not only data of an object but also methods of how to access data. With all the data stored in the form of objects, no misplacing or interference of data on the blackboard occurs. Only certain specified methods have access to data on the blackboard. Thus, using a separate database management system for the blackboard can be avoided.

7.5 Implementation

We present the implementation of our model in C++ [Stroustrup, 1991] on a simulated concurrent processing platform. There are two ways to simulate concurrency in C++. One is the application of the task library developed by AT&T [1989] as a supporting library to C++. Another one is the application of system function calls on UNIX operating system. The task library is also called co-routine library. It is composed of primary basic classes: tasks, schedulers, queues, timers, histograms, and interrupt handlers, and two base classes: object and sched (base class for schedulers). The task library is a good toolbox for concurrency simulation, but it has many restrictions on user-defined classes. For the sake of generality and portability, we use the UNIX operating system functions directly in this investigation. Our COOP model is implemented on a Sun SPARC workstation using the Sun operating system 4.1.1.
We review briefly those techniques used to establish a concurrent object-oriented programming environment under UNIX operating system. For additional methods on UNIX system programming refer to Haviland and Salama [1987], and on concurrent programming refer to Adeli [1992a & b] and Snow [1992].

Concurrent Execution:

A new process is generated by calling the system function \texttt{fork()} as shown in Figure 62. Each \texttt{fork()} call returns two process identifiers (pid). If pid is zero, the process is called new process or child process. If it is positive, the process is called original process or parent process. Which process, the original process or new process, is executed first is not defined. This matches the actual situation in multiprocessor computers.

```c
int pid= fork();
if(pid== 0)
{
  //child process;
  execl ("executable file", (char*)0);
}
else
{
  // parent process
  wait (0);
  ....
}
```

Figure 62. Concurrent Execution
Figure 63. Calling system function `fork()`
(a) thread generation  (b) without `exec()` function  (c) with `exec()` function
A thread in concurrent programming is created by a fork as the starting point and a join as the ending point (Figure 63 (a)). UNIX has no join function, but the function wait() can force the original process to wait until the end of the new process. Thus, the join synchronization can be achieved by calling the wait() function. Repetitive calling of the fork() function generates multiple processes. We use this technique to create the required processes for our simulation.

In general, functions fork() and exec() are called in pair. Each fork() generates a copy of the original process. All operations between the join and the end of the program are executed twice, once in the original process and another time in the new process. For example, there are three operations in Figure 63 (a). Operations F1 and F2 are to be executed in two processes, and operation F3 in either one of them. However, two threads generated by fork() do not combine together after join. Two threads continue the execution on their own tracks: the operation F3 in Figure 63 (a) is executed in both new and original processes (Figure 63 (b)). This redundancy can be avoided by executing operation F2 in a new program. The new program is executed in the new process by calling exec() function. Thus, F3 is executed in the original process only (Figure 63 (c)). From this point of view, the UNIX fork() system function generates a program level concurrency (two programs running concurrently) in contrast to procedure level concurrency (two procedures within one program running concurrently). In other words, the operation to be run concurrently has to be compiled as an individual program, and to be executed as in exec("executable file", (char*)0). The inconvenience of using exec() function is that data cannot be passed as parameters as in the conventional function calls. Interprocess communication is necessary. For example, if the results from F2 is desirable at the end of the original process, a message needs to be passed from the new process to the original process through a pipe between the two processes (Figure 63 (c)).
Semaphore:

Semaphore is a synchronization algorithm proposed originally by Dijkstra [1965]. UNIX operating system provides a set of semaphores by calling the semaphore control function `semget()` to create a set of semaphores and obtain the semaphore identifier (semid). There are two more system functions related to the semaphore function: `semctl()` (semaphore control) and `semap()` (semaphore operation). The semaphore related system functions are rather difficult to understand and use. We create a semaphore class `SEMAPHORE` to simplify the user interface for the convenience of applications. The class card of `SEMAPHORE` is shown in Figure 64. A class card is an index card to list the primary data and method names of a class and the position of the class in the class hierarchy tree [Yu and Adeli, 1991].

Our primary concern in design of the `SEMAPHORE` class is sharing semaphore variables in different processes. For the synchronization purpose, semaphore operations `wait()` and `signal()` should work on the same semaphore variable. However, after each `fork()` system call to generate a new process, the process will have its own copy of the `SEMAPHORE` instance. In other words, each semaphore works independently in individual processes. Thus, we create a shared memory for semaphore variables. That is why private data "shmkey" (shared memory key) and "shmid" (shared memory identifier) are included in the class definition. Methods shown in the class card are not discussed in detail for the sake of brevity.

Pipe:

Pipe is the simplest communication channel between two processes. Messages are passed through a pipe. The code in Figure 65 shows how to open a pipe for two processes. The operating system assigns a read file descriptor and a write file descriptor in a two-dimensional array `p[2]` through the system function `pipe()`. These two file
<table>
<thead>
<tr>
<th>Class Name</th>
<th>SEMAPHORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Class</td>
<td></td>
</tr>
<tr>
<td>Composed Class</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>key_t semkey, shmkey;</td>
</tr>
<tr>
<td></td>
<td>int semid, num_sem, shmid;</td>
</tr>
<tr>
<td></td>
<td>int process_count, *count;</td>
</tr>
<tr>
<td></td>
<td>struct sembuf *p;</td>
</tr>
<tr>
<td></td>
<td>struct sembuf *v;</td>
</tr>
</tbody>
</table>

**Method**

**public:**

- SEMAPHORE (key_t, key_t, int, int)
- ~SEMAPHORE ()

- int init_sem ()
- int set_sem_val (int, int):
- int set_bi_sem (int):
- void def_sem_all (short, short, short):
- void def_sem_all (ushort*);
- void def_sem_p (int, short, short):
- void def_sem_v (int, short, short):
- void wait (int);
- void signal (init);
- void report (char*);
- int get_semid();
- int get_val (int);
- int get_proc_ID ();
- int get_NCNT (int);
- int get_ZCNT (int);

**protected:**

- int remove ();

---

Figure 64. Class card of class SEMAPHORE
descriptors can be inherited by two processes beyond `fork()` shown in Figure 66. The pipe opened in an upper level is shared by all processes in lower levels. However, the opposite is not true. If one process both reads and writes on one pipe at the same time, it causes confusion and may cause error. To circumvent this problem, we use one unidirectional pipe in each process. If the pipe is for reading the message, the write file descriptor is closed; similarly, if it is for writing the message, the read file descriptor is closed. This reduces the number of file descriptors one process has. If one process has both read and write functions, two pipes need to be opened.

```c
int p[2];
int data= 123;
// open pipe
if(pipe (p)<0)
    cout< "Pipe opening failure"<<endl;
if(fork () ==0)
    {  
close (p[0]); // close reading end
    write (p[1], data, sizeof(data));
    }
else
    {
close (p[1]); // close writing end
    wait (0)
    read (p[0], data, sizeof(data));
    }
```

Figure 65. An example of creating a pipe
The communication tool, pipe, as introduced above has two deficiencies: first, two processes communicating with a pipe must have the same ancestor process; second, the pipe is not kept open during the entire program execution period. We can overcome these difficulties by using named pipes. A named pipe has a name, indicating its owner and a size similar to a regular file. It can be opened and closed at the shell command level. The UNIX operating system command `mknod` is used for creating a named pipe on the system. The file descriptor is obtained as follows:

```c
file_des= open ("named_pipe_name", O_RDONLY);
```
This is similar to opening regular files. O_RDONLY is an access permission flag (read only in this case). Other operations of named pipes are the same as conventional pipes.

**Message Queue:**

Message queue is another method for IPC. The advantages of message queues over pipes are: no first-in-first-out (FIFO) restriction, and contents of the queue can be selected with different orders. An example of message passing via a queue is shown in Figure 67. First, we open a message queue by calling **msgget()** to obtain a message queue identifier (msg_qid). The key value ((key_t)0111) has the same function as a file name to distinguish different queues. The number 0660 stands for access permission of the queue: owner reads/sends only. Two flags IPC_CREAT and IPC_EXECL stand for creating a new queue.

The difference between messages sent to the queue and those sent to the pipe is that the former message is in the form of user supplied template. The message data structure is composed of two components: a long integer variable **mtype** and an array of messages **mtext**. The extra data **mtype** is included for selective reading purposes. It is a tag to the message for identification. The data type and length of **mtext** have no restrictions at all. The actual size of the message being sent is indicated during **msgrcv()** function call. The flag IPC_NOWAIT means if the message sending fails, return -1 immediately. The message on the queue is removed after it has been read. The fourth parameter in the function **msgrcv()** indicates which message to be received from the queue. The value -1 means selection of a message from the lowest mtype value. After the queue has been used, it is removed from the computer memory by system call **msgctl()** with IPC_RMID as the flag.
Shared Memory:

In shared memory no physical data transfer occurs. Data used in different processes share the same memory address on the computer. The operations on a shared memory are similar to a message queue, except obtaining the memory address of data is the primary focus. The code in Figure 68 shows an example of its usage. To share the data, the memory segment of data should be attached to its logic data space using the `shmat()` system function. The counterpart of `shmat()` is `shmdt()` to detach from the shared memory. Detaching from the shared memory is important, since there is a limitation on how much data can be attached to one shared memory address. After its task is accomplished, the shared memory is removed from the system by calling `shmctl()` with IPC_RMID as the flag.
7.6 Applications

In this section, we present a case study of the proposed model for finite element analysis (FEA) of a composite laminated structure. Composite laminates are being used increasingly in space vehicles, commercial aircraft, automobiles, and so on. Figure 69 shows a composite laminate sheet. It has five plies. Each ply has different materials and fiber orientation. The composition of the plate makes its analysis complicated. We use the global-local finite element modeling approach to conduct stress analysis of composite laminates [Pagano and Soni, 1983] and [Chaturvedi et al., 1992]. The global-local model has the advantage of decreasing the number of unknown variables in the analysis. Accordingly, the laminate is divided into two regions along the z direction. One is the
global region, and another one is the local region. In the global region, several laminae are modeled as a unified lamina in order to decrease the number of unknown variables. In the local region, each lamina is modeled separately with the objective of obtaining detailed information. The finite element mesh is generated in the x-y plane.

Figure 69. A composite laminated structure
We recently developed an object-oriented finite element (OOFEA) class library [Yu and Adeli, 1992 and 1993] to assist object-oriented finite element analysis of structures. Considering the quantity and complexity of data involved in the finite element analysis, object-oriented data management techniques are employed in our OOFEA model. The OOFEA class library has the following classes: VECTOR, MATRIX, GENERAL_DATA, SHAPE, JACOB, NODE, GAUSS, ELEMENT_SEQ, ELEMENT, and ELEMENT_TYPE. Some of these classes, for example SHAPE, NODE and ELEMENT, are virtual classes with basic data and methods, so that users can derive their own classes from them. Problem-specific data and methods can be defined in derived classes. The OOFEA class library has been used in the implementation of our COOP model.

Software System Design Based on the Proposed COOP Model:

The major steps involved in the finite element analysis are: inputting the structure and loading information data, constructing the element stiffness matrices, solving linear simultaneous equations in order to find unknown variables, and computing the stresses and strains of the structure as the output. Each step is a task and can be executed separately. We discuss our concurrent programming approach to two of these tasks, **Input** and **Stiff**.

**Task Input**

There are six sub-tasks in the task Input. Task General_Data takes care of general information about FEA, such as number of nodes, number of elements, number of materials, and so on. Task Node records coordinates of each node. Task Material records mechanical properties and fiber orientation of composite materials. Task Node_Seq records the nodal sequence in each element, and different element types. Task
Load records load information, including mechanical load and temperature variation. Task BC records boundary conditions of the problem.

Among these tasks, task General_Data has to be executed first in order to obtain all the necessary information about the analysis. Remaining tasks are at the same level in terms of execution priority. In particular, tasks Material, Load and BC are independent. They can be executed in parallel without synchronization and IPC.

Figure 70 shows the synchronization graph for task Input. The synchronization graph we developed has the following characteristics: i) tasks are presented in parallel; ii) if two tasks need to be synchronized, the semaphore sign appears between them; iii) a semaphore sign attached to the task sign shows mutual exclusion being applied to this task; iv) pipes, message queues, and shared memory are connected to each individual task whichever is applicable; and v) the blackboards in Figure 61 are implemented by a series of databases, messages for IPC, and shared memory variables.

In our OOFEA model, two data objects act as key parameters: object GD of class GENERAL_DATA and object ovm of class OVM (object version manager). As stated earlier, GD holds data of general description about the problem. Object ovm keeps version information about all data stored in database DB1 in terms of data type, length, offset on the file, version and so on. The ovm information is stored separately in database DB2, which is part of the global blackboard.

Data in objects ovm and GD are shared among all tasks so that any changes made in one task will be reflected in other tasks. The shared memory identifiers are sent to each process through named pipes. Use of the named pipe is convenient for independent processes. The file identifier of a named pipe can be obtained easily as long as the pipe name is pre-defined in all processes.
Figure 70. Synchronization graph of task Input
Mutual exclusion is especially important in interactive input operations. Although we cannot predict which process is executed first, as soon as it is executed, we should prevent any disturbance by other tasks. If this protection measure is not taken, when a task pauses to receive input, another task starts execution and asks for input. The user of the software system will be confused, or the input from the user may be placed in a wrong processor.

In the task Node_Seq, the element geometry information is necessary to determine the element type in the finite element analysis (Figure 71). This requires nodal coordinates to be sent from the task Node to the task Node_Seq. However, when task Node_Seq on one processor requests nodal information from the task Node on another processor, some nodes may not be available yet. We cannot send nodal coordinates via pipe continuously as soon as they are inputted. Therefore, we use the message queue for this purpose. A node data structure is defined with nodal number as \textit{mtype} and 2-D coordinates as \textit{mexr} (refer to previous section about message queue parameters in the UNIX operating system). Each object of the node data structure corresponds to one node. It is picked up from the queue in the task Node_Seq according to its node number. In addition, we have to consider that some nodes are used in more than one element, but message queue removes any information from the queue as soon as it is read. Several copies of the same node are necessary in some cases. That is why there is a synchronization semaphore between the tasks Node and Node_Seq to ensure that the frequency of each node's appearance is counted at first and then nodal coordinates are sent to the queue in multiple copies. The shared object \textit{GD} helps to transfer the updated node copy number from the task Node_Seq to the task Node in time.

File I/O (input and output) is one way of transferring information between processes. However, the file I/O process involves mechanical movement of file/disk head and is far slower than computer memory access. For this reason, we transfer all the
necessary information among the processes in each version through either one of the following methods: pipes, message queues, and shared memories. Files are not suitable as information exchange media among processes, contrary to their frequent use in sequential programming. When computation of one version is completed, all data about this version are stored into a database. Some or part of those data can be retrieved from database if they are used in another version of computations. Thus, re-inputting the archived data is avoided.

![Diagram](image)

**Figure 71. Synchronization graph of task Stiff**
**Task Stiff**

As mentioned earlier, in our OOFEA approach, elements are categorized into different types. Instead of computing the properties of all the finite elements in the task Stiff, only the properties of a representative number of elements are computed, because the elements with the same geometrical shape and material properties have the same stiffness matrix. The element types in the problem domain are determined in the task `Node_Seq`. Each element records its type identifier. The total number of elements is recorded in the object `GD`. During the assembly process of the global stiffness matrix, each element is related to its element type and the corresponding element stiffness matrix is used for assembling the global stiffness matrix.

Within the task Stiff, all element stiffness matrices $[K_1], [K_2], ..., [K_n]$ are calculated by sub-tasks $E_1, E_2, ..., E_n$ concurrently (Figure 71). Task $E_i$ computes the stiffness matrix of element $i$. For the convenience of discussion, these tasks are collectively referred to as the task `Element`. Task Stiff acts as a global task to control element stiffness matrix computations by local tasks and generates enough processes for them. However, no one process is assigned specifically to one `Element` task. Element type identifiers are posted on the blackboard. Any process can read an element number from the blackboard. After one process reads an element number, this number is removed from the blackboard. Other processes can only take the remaining numbers. The order of process execution is random. Two semaphores are used for synchronization in the task Stiff, since it requires information from the task `Input` and has the responsibility of notifying the task `Solver` to solve the linear simultaneous equations.

Element stiffness matrices are the product of the task `Element`. They are stored in the database `DB1` directly for the task `Solver`. The offset of each stiffness matrix in the database `DB1` is recorded in the object `ovm`. One issue in concurrent programming is the simultaneous access to one shared data source on the computer. Blocking the access to
the file (called file locking) is the common solution to the problem. Since the size of
each element stiffness matrix and its element type identifier are known to each element
object, the offset of each element stiffness matrix object can be calculated before storing
the matrix on the file. Two stiffness matrices of different element types will not be
recorded at the same position on the file. Therefore, file locking is not necessary in our
approach. This saves waiting time for file unlocking during the storage of element
stiffness matrices.

The task Element computes element stiffness matrices. Figure 72 shows a
synchronization graph for the task E1 of Figure 71. The information about general data
and version is shared among all processes. Computation of shape functions and Jacobian
matrix is not related to version at all. The material information is transferred from the
task Input via a named pipe. The composite material stiffness and compliance matrices
are computed once in one element and shared among all processes. The assembly of the
stiffness matrix of each layer to element stiffness matrix depends on the position of the
lamina in the laminate. They are added to the different blocks in the element stiffness
matrix [Chaturvedi et al., 1992]. This element stiffness matrix eventually contributes to
[K1] in Figure 71.
7.7 Performance Evaluation

The goal of performing parallel processing is to speed up the computation. Speedup is a common evaluation index in parallel processing. Because different processing times are involved, and the evaluation of those times is machine dependent, various definitions of speedup can be found in the literature [Sun and Ni, 1990]. In general, the speedup is defined as the ratio of elapsed time of parallel computation on one processor to that on n processors execution in parallel. The elapsed time consists of synchronization time, inter-process communication time, concurrent execution time, and
so on. In this research, we have simulated concurrent processing on a uni-processor computer. UNIX operating system has a system function times() that provides the following four CPU times: user time of original process, system time of original process, total user time of all new processes, and total system time of all new processes. The user time is the CPU time used for executing instructions in the program. The system time is the CPU time used by the calling process. The summation of user and system CPU time covers all elapsed time except synchronization time for unevenly allocated process. If we assume that this synchronization time is negligible (perfect parallelism), then the elapsed time is equal to the summation of user and system CPU times. We use this elapsed time for computing speedup. Because we create new processes to compute element stiffness matrices, the elapsed time is taken from the new process part of times() output.

To have a clear picture about CPU time required for both concurrent execution and IPC, we ran a test example on our machine (Sun SPARC workstation under operating system 4.1.1). Our test includes concurrent execution with fork() and exec() functions, attaching to and detaching from the shared memory of an object, and sending and receiving an object through a named pipe. The test results clearly show that execution of concurrent processes dominates a large portion of overhead time in our simulation. It takes about eighteen times more CPU time than that for IPC. By comparison, we learned that attaching to and detaching from the shared memory require about 2/3 of the user CPU time and 1/7 of the system CPU time of message passing through named pipes.

In our parallel programming approach described above, both function and processing time are of primary concern in determining the appropriate IPC method. For example, we design the objects GD and ovn to be shared by all processes. This requires additional operative time for attaching to and detaching from the shared memory in addition to message passing (notifying shared memory address through named pipes).
The advantage is that all changes made on these objects in one process are distributed to other processes simultaneously and shared by all processes. If these objects are not shared, after each process is completed, the modified GD and ovm objects need to be passed back to where they came from, and then to be re-sent to other processes for updating. This procedure requires more computing time, because message passing takes more CPU time than using shared memory as our test example shows.

The concurrent processing simulation results of our COOP model are shown in Figures 77 and 78. We use eight finite elements and 1, 2, 4, and 8 processors. These figures show the results for two different runs. Since the machine is used in a time-sharing mode, the accumulated errors in CPU time computation varies from run to run. This causes minor variation in the speedup as indicated in Figures 77 and 78.

Figure 73 shows the speedup for a mesh with eight finite elements using 1, 2, 4, and 8 processors. The derivation from the ideal line in Figure 73 is due to overhead time required in IPC and initiation of concurrent processes. In spite of this, a speedup of over 90% is achieved using eight processors. Figure 74 shows the variation of user and system CPU times in the test. As expected, user CPU time varies little, while system CPU time increases substantially with the number of processors.

7.8 Closure

This chapter presents a concurrent object-oriented programming model using a blackboard architecture. The evaluation results show that this approach reaches two goals of developing software systems in computer-aided engineering, ease of maintenance and high execution time. In addition, the simulation of concurrent programming on a uniprocessor computer is valuable in the absence of real parallel computers for verifying a new model.
Figure 73. Speedup
Figure 74. Variations of user and system CPU times
8.1 Introduction

Over the last two decades, researchers have proposed many algorithms for solving the following system of linear equations using both sequential and parallel computers:

\[ A \mathbf{X} = \mathbf{B} \]  

(81)

where \( A \) is a symmetric and positive definite matrix. A large system of linear equations usually includes thousands of unknown variables in the vector \( \mathbf{X} \). Much research has been done using the Gaussian elimination method to solve Eq. (81) in a parallel processing environment. Heath et al. [1991] present a review of parallel algorithms for sparse linear systems of equations. These algorithms focus on the following solution phases: ordering, symbolic factorization, numerical factorization, and triangular solution.

Object-oriented programming (OOP) has gained popularity in recent years for the development of engineering software systems. Its characteristics of inheritance, polymorphism, modularity, information hiding, and data abstraction contribute to this popularity [Yu and Adeli, 1991]. OOP has been used in expert systems [Forde and Stiemer, 1989, Adeli and Hung, 1990], integrated design systems [Abdalla and Yoon, 1992, Ahmed et al., 1992, Nakai et al., 1992], and finite element analysis [Baugh, 1989, Forde and Stiemer, 1989, Yu and Adeli, 1993]. A number of papers can be cited on the application of OOP for solution of a system of linear equations. McDonald [1989]

We point out the shortcomings and limitations of the object-oriented equation solvers reported in the literature. First, to emphasize the class hierarchy in their class libraries, many abstract classes are created to allow for the completeness of mathematical concepts. These abstract classes are rarely used in any application. The class most often used in matrix operations is the simple class at the bottom of such complicated class hierarchy.

Second, the research reported in the literature has focused on the operations within a matrix, such as the arrangement of matrix elements, and computation sequences to reduce round-up errors or to avoid ill-conditioning of the coefficient matrix in Gaussian elimination. Because such operations are performed at a lower level, that is by one method within an object, we can take little advantage of the unique features of OOP.

The third and primary issue: what is the most effective approach for solving systems of linear equations using the OOP paradigm in a concurrent processing environment? In conventional programming, Gaussian elimination method has become the most popular approach for solving systems of linear equations because of its ease of implementation and its efficiency when used on conventional uniprocessor machines. With the increasing availability of high-performance computers, it is time to re-examine the indirect iterative methods.
Iterative methods are attractive for solving systems of sparse linear equations in a concurrent processing environment for the following reasons: 1) the convergence of iterative methods for solution of systems of linear equations is guaranteed if the coefficient matrix $A$ is symmetric and positive definite, which is the case in a majority of engineering problems; 2) iterative methods are easier to implement than direct solution methods, such as the Gaussian elimination method, because the repetitive vector product operation is the only major operation in the entire process; 3) recognizing the sparseness of the coefficient matrix in many CAE applications such as the structural analysis problem, creating a sparse vector object class from an existing regular vector object class is a good example of taking advantage of OOP features which will be demonstrated in a subsequent section; 4) considering the static nature of nonzero elements in the coefficient matrix $A$, the interprocess communication (IPC) requirement is lower compared with other direct solution methods in a parallel computing environment; and 5) the sparseness of the coefficient matrix reduces the CPU time significantly since computations on zero elements are avoided.

In the following sections, we will begin by introducing the iterative methods for the solution of linear equations. Then, we will analyze the nature of iterative methods and will design a sparse vector class based on a previously developed full vector class in our finite element analysis class library [Yu and Adeli, 1993], and finally we will present a new algorithm for solving a system of sparse linear equations in a concurrent object-oriented programming environment.

### 8.2 Iterative Methods For Solving Systems of Linear Equations

A system of linear simultaneous equations is represented by Eq. (81), where $A \in \mathbb{R}^{n \times n}$, $X \in \mathbb{R}^n$, $B \in \mathbb{R}^n$, and $\mathbb{R}^n$ is a space of $m$-dimensional real numbers. We
assume \( m \) is equal to \( n \) in order to simplify our discussion. The basic technique for solving Eq. (81) in iterative methods is to split matrix \( A \) into matrices \( S \) and \( T \):

\[
A = S + T \tag{82}
\]

such that Eq. (81) becomes:

\[
S X^{(r+1)} = B - T X^r \tag{83}
\]

where \( r \) refers to the iteration number.

The simplest splitting of coefficient matrix \( A \) is to separate the diagonal elements from other elements. In this case, matrix \( S \) contains diagonal elements and matrix \( T \) contains the remaining elements. Thus, the solution \( X \) of Eq. (81) can be expressed explicitly as follows:

\[
x_i^{(r+1)} = \frac{1}{A_{ii}} b_i - \sum_{j \neq i} A_{ij} x_j^{(r)}, \quad i = 1, \ldots, n \tag{84}
\]

This is the Jacobi method.

In order to minimize the required storage space, an important consideration for large systems of equations, any newly obtained vector element \( x_i^{(r+1)} \) replaces \( x_i^{(r)} \) immediately. Thus, vector \( X^{(r)} \) can be split into two parts:

\[
X^{(r)} = \begin{cases} 
  x_{k1}^{(r+1)}, & k1 = 1, \ldots, i-1 \\
  x_i^{(r)}, & k2 = i + 1, \ldots, n
\end{cases} \tag{85}
\]

The newly obtained elements of \( X^{(r)} \), denoted by \( x_{k1}^{(r+1)} \), are used immediately to calculate the remaining elements of \( X^{(r)} \), denoted by \( x_i^{(r)} \). The solution \( X^{(r)} \) of Eq. (81) can be expressed as:

\[
x_i^{(r+1)} = \frac{1}{A_{ii}} b_i - \sum_{j \neq i} A_{ij} x_j^{(r+1)} - \sum_{j > i} A_{ij} x_j^{(r)}, \quad i = 1, \ldots, n \tag{86}
\]

This is the Gauss-Seidel method.
To improve the convergence of iterative methods, we can take a correction vector $e$ into consideration during the computation of $X^{(r+1)}$:

$$X^{(r+1)} = X^{(r)} + \omega e \tag{87}$$

where the correction vector $e$ is defined as

$$e = X^{(r+1)} - X^{(r)} \tag{88}$$

Note that the $X^{(r+1)}$ term in Eq. (88) is calculated from Eq. (86), while the $X^{(r+1)}$ term in Eq. (87) is the improved $X$. Since this is the extrapolation or over relaxation of the or, it is called the Successive Over Relaxation (SOR) method. The parameter $\omega$ is therefore called the relaxation factor. The solution $X$ of Eq. (81) can now be expressed as:

$$x_i^{(r+1)} = x_i^{(r)} + \omega \left[ \frac{1}{A_{ii}} b_i - \sum_{j=1}^{i-1} \frac{A_{ij}}{A_{ii}} x_j^{(r)} - \sum_{j=i+1}^{n} \frac{A_{ij}}{A_{ii}} x_j^{(r)} - x_i^{(r)} \right], \quad i = 1, \ldots, n \tag{89}$$

Note that Eq. (89) becomes identical to Eq. (86) when $\omega = 1$. That is, the Gauss-Seidel method is a special case of the SOR method when $\omega$ equals 1. Furthermore, if the newly obtained vector element $x_i^{(r+1)}$ does not replace $x_i^{(r)}$, Eq. (89) becomes the same with Eq. (84) (the Jacobi method) when $\omega$ equals 1. The convergence requirements for the coefficient matrix $A$ are different in the three methods of Jacobi, Gauss-Seidel, and SOR. A detailed description of these requirements and how to choose an optimum $\omega$ can be found in Strang [1988] and Schendel [1989]. We assume the system of linear equations converges to its solution $X$ in our discussion.

### 8.3 Analysis of The Characteristics Of Iterative Methods

To explore some interesting characteristics of iterative methods, we take a close look at a system of linear equations with four unknown variables:
Out of the three aforementioned iterative methods, for the sake of simplicity, we apply the Jacobi method to Eq. (90) and obtain:

\[
\begin{bmatrix}
 a_{11} & a_{12} & a_{13} & a_{14} \\
 a_{21} & a_{22} & a_{23} & a_{24} \\
 a_{31} & a_{32} & a_{33} & a_{34} \\
 a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix}
\begin{bmatrix}
 x_1 \\
 x_2 \\
 x_3 \\
 x_4
\end{bmatrix}
= 
\begin{bmatrix}
 b_1 \\
 b_2 \\
 b_3 \\
 b_4
\end{bmatrix}
\]  

(90)

This equation can be rewritten as:

\[
\begin{bmatrix}
 x_1 \\
 x_2 \\
 x_3 \\
 x_4
\end{bmatrix}^{(r+1)}
= 
\begin{bmatrix}
 \bar{b}_1 \\
 \bar{b}_2 \\
 \bar{b}_3 \\
 \bar{b}_4
\end{bmatrix}
- 
\begin{bmatrix}
 \bar{a}_{11} & \bar{a}_{12} & \bar{a}_{13} & \bar{a}_{14} \\
 \bar{a}_{21} & \bar{a}_{22} & \bar{a}_{23} & \bar{a}_{24} \\
 \bar{a}_{31} & \bar{a}_{32} & \bar{a}_{33} & \bar{a}_{34} \\
 \bar{a}_{41} & \bar{a}_{42} & \bar{a}_{43} & \bar{a}_{44}
\end{bmatrix}
\begin{bmatrix}
 x_1 \\
 x_2 \\
 x_3 \\
 x_4
\end{bmatrix}^{(r)}
\]  

(91)

where

\[
\bar{b}_i = \frac{b_i}{a_{ii}}, \quad i = 1, \ldots, 4
\]

(93)

\[
\bar{a}_{ij} = \begin{cases} 
\frac{a_{ij}}{a_{ii}}, & i, j = 1, \ldots, 4, i \neq j \\
0, & i = j
\end{cases}
\]

(94)

Eq. (92) shows clearly that the iterative method primarily involves a vector multiplication and a simple number addition to calculate each \( x_i \) in iteration \((r+1)\) as follows:

\[
x_i^{(r+1)} = \bar{b}_i - \bar{A}x^{(r)}
\]

(95)
where $\overline{A}_i$ is the $i$-th row vector $[\overline{a}_{i1}, \overline{a}_{i2}, \overline{a}_{i3}, \overline{a}_{i4}]$ of the coefficient matrix in Eq. (92).

This simple analysis leads us to delineate some characteristics of iterative methods for solving a system of linear equations:

1) After splitting the matrix $A$ in Eq. (90) into the matrix $S$ (containing only the $A$'s diagonal elements) and the matrix $T$ (containing the remaining elements of $A$), the resulting vectors $\overline{A}_i$ in Eq. (95) do not undergo any change during the entire iteration process.

2) Since the only operations in Eq. (92) are matrix splitting and division of the elements of the coefficient matrix by one nonzero diagonal element on the same row (note the definition of $\overline{b}_i$ and $\overline{a}_{ij}$), zero elements remain zero. This static nature of zero elements is exploited in developing an algorithm for solution of a system of sparse linear equations.

3) Vector multiplication is the major computation in Eq. (95).

4) Computation of $\overline{b}_i$ and $\overline{A}_i$ is done independently of each other. As long as $\overline{A}_i$ vectors are multiplied by the same $X^{(r)}$, each $\overline{A}_i$ can be stored in a different processor in the parallel computing environment. No interprocess communication is necessary.

5) The diagonal elements $\overline{a}_{ii}$ in the modified coefficient matrix of Eq. (92) are all zero. This further reduces the memory requirement for the coefficient matrix. In fact, except for the purpose of recovering the original coefficient matrix, the diagonal elements are no longer needed after Eq. (92) is formed.

6) In addition to zeroes in the diagonal elements in Eq. (92), there are additional zeroes in other elements of a large sparse coefficient matrix. Since the multiplication of a row vector of the modified coefficient matrix with vector $X^{(r)}$ is the sum of the multiplication of correspondent elements of two vectors, the multiplication by zero elements does not contribute to the final result, and therefore should be avoided.
In summary, we should take advantage of sparseness, static nature of zero terms in the 
coefficient matrix, and independent vector product operation in developing an efficient 
linear equation solver. The above observations are true to SOR iterative method either.

8.4 Applying OOP to Sparse Vector Operations

An important feature of a system of linear equations encountered in many 
engineering problems is that the coefficient matrix $A$ in Eq. (81) is a sparse matrix with 
many zero elements. For the sake of brevity, we discuss a simple example of a two-
dimensional finite element grid with N elements (N being an even number) and 1.5N+3 
nodes shown in Figure 75. Node numbering is done such that the bandwidth is 
minimized. The bandwidth is six in this case. With this node numbering, there are only 
three nonzero elements among 1.5N+3 elements in a row vector of matrix $A$. When the 
number of elements N increases from 10 to 100 and 1000, for example, the nonzero ratio 
$R_n$ (the ratio of the number of nonzero elements to all elements) of each row vector 
decreases from 16.67% to 3.27% and 1.96%. An efficient vector operation needs to be 
developed for such sparse linear systems of equations in order to minimize the computer 
processing time and the memory requirement.

The size of the computer’s memory limits the number of elements of matrix $A$ that 
can be stored in-core. The frontal method and other solution algorithms maintain only 
part of matrix $A$ in the computer main memory at any one time. In our approach, 
however, we take full advantage of the sparseness of matrix $A$ by storing only the 
nonzero elements, the value of each nonzero element, and an index identifying its 
location. The zero elements are not stored and no computation is performed on them.
We treat a sparse matrix as a set of sparse row vectors. Because each row vector has at least one nonzero element, all row vectors must be stored. The storage memory we can save is within the row vectors. In addition, the discussion in the previous section indicates that vector product operation is the primary operation in iterative methods. Therefore, we will concentrate on studying the sparse row vectors instead of the sparse matrices in our discussion. The application of sparse vector operation developed in this section to the sparse matrix will be shown in the following section.

We exploit two particular features of OOP: the reuse of existing codes by class inheritance relationships and member overloading (which refers to using the same method name for different operations in various methods). We build our \texttt{SPARSE VECTOR} class on the basis of a full \texttt{VECTOR} class. Figure 76 shows the class card of our full \texttt{VECTOR} class, developed as part of our research on object-
oriented finite element analysis [Yu and Adeli, 1993]. The VECTOR class has one double float number string called Array to represent all elements in a vector. The size of the vector is defined by integer number Dim. By applying the inheritance feature of object-oriented programming, we create a derived class SPARSE_VECTOR from the base class VECTOR, as shown in Figure 77. The differences between SPARSE_VECTOR and VECTOR classes are:

1) In addition to a double float number string Array, a new string Nonzero_Index is defined to record the nonzero information for the sparse vector;

2) The integer number Dim indicates the number of nonzero elements, while its original function in class VECTOR is replaced by integer number Size in class SPARSE_VECTOR. The reason is that only the data array with dimension Dim takes part in the vector multiplication.

3) The length of strings Array and Nonzero_Index is determined by the number of nonzero elements in order to avoid any memory waste in storing zero value elements. This is where we decrease the memory requirement for a sparse vector; and

4) Many methods in VECTOR are either reused or redefined (but carry the same name) to meet the new data type requirements.
<table>
<thead>
<tr>
<th>Class Name</th>
<th>VECTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Class</td>
<td></td>
</tr>
<tr>
<td>Composed Class</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>int Dim;</td>
</tr>
<tr>
<td>double *Array;</td>
</tr>
<tr>
<td>char *Name;</td>
</tr>
<tr>
<td>short ShareVector;</td>
</tr>
<tr>
<td>Object_type Type:</td>
</tr>
<tr>
<td>Bool Active;</td>
</tr>
<tr>
<td>int form;</td>
</tr>
<tr>
<td>char* File_name;</td>
</tr>
<tr>
<td>streampos Offset, Length;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected:</td>
</tr>
<tr>
<td>VECTOR&amp; SetUpO;</td>
</tr>
<tr>
<td>VECTOR&amp; Copy(const VECTOR&amp;. int);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>public:</th>
</tr>
</thead>
<tbody>
<tr>
<td>VECTOR();</td>
</tr>
<tr>
<td>VECTOR(int);</td>
</tr>
<tr>
<td>VECTOR(int, double ...);</td>
</tr>
<tr>
<td>VECTOR(int, double *);</td>
</tr>
<tr>
<td>VECTOR(const VECTOR&amp;);</td>
</tr>
<tr>
<td>VECTOR(const VECTOR&amp;, int);</td>
</tr>
<tr>
<td>-VECTOR();</td>
</tr>
<tr>
<td>VECTOR&amp; SetUp(int);</td>
</tr>
<tr>
<td>VECTOR&amp; SetUp(int, int);</td>
</tr>
<tr>
<td>Bool Store(char *, streampos);</td>
</tr>
<tr>
<td>Bool Retrieve(char *, streampos);</td>
</tr>
<tr>
<td>Bool Delete(char *, streampos);</td>
</tr>
<tr>
<td>streampos Insert(char *, streampos);</td>
</tr>
<tr>
<td>Bool Update(char *, streampos, int, double);</td>
</tr>
<tr>
<td>double Retrieve(char*, streampos, int);</td>
</tr>
<tr>
<td>double&amp; operator <a href="int"> </a> const;</td>
</tr>
<tr>
<td>VECTOR&amp; operator +=(const VECTOR&amp;);</td>
</tr>
<tr>
<td>VECTOR&amp; operator -==(const VECTOR&amp;);</td>
</tr>
<tr>
<td>VECTOR&amp; operator *==(const VECTOR&amp;);</td>
</tr>
<tr>
<td>VECTOR&amp; operator +==(const VECTOR&amp;);</td>
</tr>
<tr>
<td>friend VECTOR operator + (const VECTOR&amp;, const VECTOR&amp;);</td>
</tr>
<tr>
<td>friend VECTOR operator - (const VECTOR&amp;, const VECTOR&amp;);</td>
</tr>
<tr>
<td>friend double operator * (const VECTOR&amp;, const VECTOR&amp;);</td>
</tr>
<tr>
<td>friend ostream&amp; operator &lt;&lt; (ostream&amp;, const VECTOR&amp;);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Non-member Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>int operator == (const VECTOR&amp;, const VECTOR&amp;);</td>
</tr>
<tr>
<td>int operator != (const VECTOR&amp;, const VECTOR&amp;);</td>
</tr>
<tr>
<td>VECTOR&amp; operator * (const double&amp;, const VECTOR&amp;);</td>
</tr>
<tr>
<td>VECTOR&amp; operator + (const VECTOR&amp;, const VECTOR&amp;);</td>
</tr>
<tr>
<td>VECTOR&amp; operator - (const VECTOR&amp;, const VECTOR&amp;);</td>
</tr>
<tr>
<td>double operator * (const VECTOR&amp;, const VECTOR&amp;);</td>
</tr>
</tbody>
</table>

Figure 76 Class card of a full vector class VECTOR
<table>
<thead>
<tr>
<th>Class Name</th>
<th>SPARSE VECTOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Super Class</td>
<td>VECTOR</td>
</tr>
<tr>
<td>Composed Class</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td>short *Nonzero_Index; int Size;</td>
</tr>
</tbody>
</table>

**Method**

```cpp
public:
SPARSE_VECTOR();
SPARSE_VECTOR(int, int, short*);
SPARSE_VECTOR(const SPARSE_VECTOR&, int);
~SPARSE_VECTOR();
SPARSE_VECTOR& SetUp(short *);
SPARSE_VECTOR& SetUp(int, int, short *);

int get_Size();
short * get_Nonzero_Index();
double& operator [] (int) const:

SPARSE_VECTOR& operator += (const SPARSE_VECTOR&);
SPARSE_VECTOR& operator -= (const SPARSE_VECTOR&);
SPARSE_VECTOR& operator *= (const SPARSE_VECTOR&);
SPARSE_VECTOR& operator = (const SPARSE_VECTOR&);

friend ostream& operator « (ostream&, const SPARSE_VECTOR&);
friend double operator * (const SPARSE_VECTOR&, const VECTOR&);
friend double operator * (const VECTOR&, const SPARSE_VECTOR&);
friend double operator * (const SPARSE_VECTOR&, const SPARSE_VECTOR&);
```

**Non-member Function**

```cpp
friend VECTOR operator + (const SPARSE_VECTOR&, const VECTOR&);
friend VECTOR operator - (const SPARSE_VECTOR&, const VECTOR&);
friend VECTOR operator + (const VECTOR&, const SPARSE_VECTOR&);
friend VECTOR operator - (const VECTOR&, const SPARSE_VECTOR&);
friend VECTOR operator + (const SPARSE_VECTOR&, const SPARSE_VECTOR&);
friend VECTOR operator - (const SPARSE_VECTOR&, const SPARSE_VECTOR&);
```

Figure 77 Class card of a sparse vector class SPARSE VECTOR
Figure 78 shows a comparison of a full vector object with a sparse vector object in terms of memory requirement. In element 1 of Figure 75, node 1 is related to nodes 2, 4, and 5. Therefore, the row vector of node 1 has only eight nonzero elements (two degrees of freedom at each node). Their indexes are 1, 2, 3, 4, 7, 8, 9, and 10. Considering the special role of diagonal element $a_{11}$ in iterative methods [Eq. (92)], only elements 2, 3, 4, 7, 8, 9, and 10 of the row vector are stored as part of the sparse matrix $A$. This is shown in the upper portion of Figure 78, where X indicates a nonzero element. In a sparse vector object, variable Dim records the number of nonzero elements, which is seven in this example. The size of the vector is recorded as 10 in variable Size. We store the index numbers of the seven nonzero elements 2, 3, 4, 7, 8, 9, and 10, in elements 1 through 7 of Nonzero_Index. The values of elements 2, 3, 4, 7, 8, 9, and 10 of the original full vector are then stored as elements 1 through 7 in the Array of the sparse vector object. A 10-element Array has been reduced to a 7-element Array in the sparse vector object.

In considering the entire memory requirement for storing both the index and values of nonzero elements, we define the ratio of memory saving $R_m$ in each row vector in our approach as follows:

$$R_m = 1 - \frac{\text{Total memory for storing a sparse vector}}{\text{Total memory for storing a full vector}}$$

$$= 1 - \frac{N_n * 8 + N_n * 2}{N * 8} = 1 - \frac{10N_n}{8N} = 1 - 1.25R_n$$

(96)

where $N, N_n,$ and $R_n$ are the number of total elements, the number of nonzero elements in a vector object, and the nonzero ratio of a sparse vector object, respectively.
Figure 78 Comparison of memory requirement for element 1 in Figure 75, full vector approach vs. sparse vector approach

Eq. (96) is based on the assumption that each double float number in Array occupies 8 bytes of memory and each short integer number in Nonzero_Index occupies 2 bytes of memory (Figures 76 and 77). Using the same example presented in the previous section (Figure 75), where the number of nodes is 1.5N+3, the nonzero ratio is 16.67%, 3.27% and 1.96% when N equals 10, 100, and 1000, respectively. Thus, we save as much as 79.16%, 95.91% and 97.55%, respectively, in memory storage compared with a full vector object. Another advantage of the sparse vector approach is that the memory saving is independent of the bandwidth. Only the nonzero elements of a vector are stored regardless of how close the values of node numbers are in one finite element. This gives more flexibility in node numbering in finite element applications.

By operating only on the nonzero elements in a row vector, the computation requirement for the vector operation decreases substantially. Therefore, the proposed
sparse vector operation using the OOP technique is efficient with regards to both memory requirement and computational speed.

It should be pointed out that the sparse vector approach requires less memory than the skyline method (Felippa, 1975] commonly used in structural analysis. The block in a skyline method begins with the diagonal element and ends at the farthest nonzero element. The memory saving is achieved by not including those elements that are beyond the farthest nonzero element within the block. This results in a partial memory saving for zero elements. For large structures with thousands of degrees of freedom, some elements within the skyline remain zero. No zero elements are stored in memory or used in computations in the sparse vector approach.

8.5 A Concurrent Object-Oriented Programming Algorithm for Solution of Large Sparse Systems of Equations

We first summarize the general strategies used in our concurrent object-oriented programming algorithm for the solution of a system of linear sparse equations in this section.

1) One of the \( N_p \) available processors is used for controlling purposes. We call it the \textit{master processor}. The remaining processors are used to compute the elements of a vector object \( X \) in \((r+1)\)st iteration. We call these \textit{slave processors}. Slave processors are assigned the index numbers corresponding to the row vector number \( i \) in Eq. (89). Each slave processor performs computation of Eq. (89) only on those rows assigned to it. The distribution of index numbers among slave processors is done as evenly as possible to balance the computing load on each processor so that the waiting period (the synchronization time) of slave processors is minimized. For example, if there are 2000 equations and 10 slave processors, each slave processor performs 200 calculations of Eq. (89).
2) Based on the discussion of the iterative methods presented previously, we note that the row vectors in coefficient matrix $A$ have no interaction with each other. This important characteristic is exploited in our concurrent algorithm. In contrast, vector object $X$ is multiplied by all the row vectors. As described in the previous section for the Jacobi method, $X$ in the $r$-th iteration is used to compute $X$ in the $(r+1)$st iteration [Eq. (84)]. In the Gauss-Seidel method, there is only one object $X$ [Eq. (86)]. New elements of object $X$ are continuously calculated and used to update the old elements. This requires vector object $X$ to be stored on the shared memory of the computer so that it can be accessed by all processors. With this arrangement, elements of object $X$ can be updated immediately and used for computing the remaining elements of $X$ when a new element is obtained in any one processor.

3) In structural analysis, a common approach is to assemble the global stiffness matrix (coefficient matrix $A$ in our discussion) before the solution of the system of linear equations. To assemble the matrix $A$, the information regarding node relationship is necessary. Our approach is to establish this relationship in the master processor and to store it in an array $Index_{node}$. This array is shared by all processors similar to the vector object $X$. Each processor assembles the necessary row vectors of the coefficient matrix $A$ based on information provided by $Index_{node}$ and stores them in its local memory. Thus, the coefficient matrix $A$ is assembled concurrently in different processors.

4) Whether to proceed to the next iteration or not is determined in the master processor. At the end of each iteration, the maximum value of the elements in the correction vector [Eq. (88)] is determined. If this value is larger than the predefined tolerance, additional iterations are necessary. If it is less than the tolerance, the computation is terminated. After all slave processors complete a given iteration, finding the maximum value of the elements in the correction vector and comparing it with the given tolerance are performed by the master processor.
5) Although computing an element of correction vector $e$ requires less time than computing an element of object $X$, because all element of $e$ are calculated in the master processor, the computing load between the master processor and the slave processor is generally balanced. Thus, the waiting time of obtaining $X_i$ from slave processors in the master processor can be ignored.

6) The role of synchronization is important in solving a system of linear equations using iterative methods. After the assigned elements of $X$ in one slave processor are calculated in the $(r+1)$st iteration, this processor has to wait until the master processor signals the $(r+2)$nd iteration. This is because the error in each iteration needs to be compared with the tolerance limit. We use named pipes, called FIFO (a communication channel between two processors to deliver information in a First-In-First-Out order), for this synchronization purpose [Adeli and Yu, 1993b]. Each slave processor has one private FIFO connected to the master processor and has access to a public FIFO. The public FIFO is used by all the slave processors and the master processor. Each private FIFO is used by one slave processor and the master processor. When a new iteration begins, the master processor signals the slave processors through the public FIFO. All slave processors receive this signal from the public FIFO. When the computation is completed, the slave processor signals the master processor through its private FIFO so that the master processor knows exactly which slave processor has completed its task.

7) The diagonal elements of the original coefficient matrix appear in the denominators of the modified coefficient matrix as shown in Eq. (91). After splitting and rearranging the original coefficient matrix, these diagonal elements are no longer needed. Therefore, they are not stored as part of any row vectors.

8) Only those elements of vector $B$ [Eq. (81)] corresponding to row vectors processed in a given slave processor are assembled and stored in that slave processor. It
is not necessary to share the entire vector $B$ by all processors. The master processor does not need it either.

9) The correction vector object $e$ in Eq. (87) is calculated and stored in the master processor. If correction vector elements are calculated in slave processors, the interprocess communication load is increased, either by sending the newly calculated elements of vector $e$ back to the master processor via a message queue or by placing the correction vector on the shared memory so that all processors can access it. Another reason for having the correction vector object in the master processor is to shift error computations from the slave processors to the master processor, thus, distributing the loads on the processors more evenly.

Figure 79 shows a concurrent computing architecture for solving sparse linear systems of equations using iterative methods. Both $X$ and $\text{Index}_\text{node}$ are stored in the shared memory. Processor 1 on the left is the master processor. The remaining processors, 2 to $N_p$, are slave processors. In a structural analysis problem, the element stiffness matrices are stored in a separate database because there is a limitation on the shared memory. All processors have access to this database for retrieving the element stiffness matrices.

It should be pointed out that in order to minimize the computations and storage requirement for element stiffness matrices, only representative element stiffness matrices are computed and stored. Elements having the same material properties and geometric shape have the same stiffness matrix.

The concurrent object-oriented algorithm for the solution of sparse linear systems of equations is presented in Figure 81. The instructions on the left and right are carried out in the master processor and slave processors, respectively.
Figure 79 Concurrent computing architecture for solving a sparse system of linear equations using iterative methods
Figure 80 Concurrent object-oriented iterative equation solving algorithm
8.6 Examples

In this section, we present three examples of the solution of linear equations generated from the structural analysis of a highrise structure shown in Figure 7. This structure is modeled as a space truss. Joints at the ground level are fixed. Loads are applied from two directions, wind load in horizontal direction and weight load in vertical direction according to the standard design code. Stress at each member and displacement at each joint occur under the applied load. Using the matrix analysis method of structural engineering, we can obtain a system of linear equations to solve the joint displacements, and therefore to compute the stress at members.

Since there is no commercial concurrent object-oriented programming language available, we implemented our algorithm in C++ programming language [Stroustrup, 1991] and simulated the concurrent computing environment on an IBM RISC/6000 workstation using UNIX system programming techniques [Adeli and Yu, 1993b].

Two types of CPU time are reported, one is the system CPU time (for IPC, process execution, etc.) and another one is the user CPU time (for executing instructions in the program). The sum of system and user CPU time is used for speedup calculations [Adeli and Yu, 1993b].

In order to evaluate the speedup of the algorithm in terms of size of the problem, we ran three sets of tests. The 30 story structure shown in Figure 81 has 2412 members and 613 nodes. The resulting 1803 linear equations are used as example 1. Examples 2 and 3 are similar to example 1, but have 5 and 70 stories, resulting in 183 and 3603 linear equations, respectively.

The test results are given in Table 1. The synchronization time is not included in our speedup calculation. This is due to the unavailability of a commercial concurrent OOP compiler, because UNIX operating system provides only the total CPU time for all slave processors.
Figure 81 A high-rise building structure
<table>
<thead>
<tr>
<th>No. of Processors</th>
<th>183 Eqs.</th>
<th>1803 Eqs.</th>
<th>3603 Eqs.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>System CPU Time (1/60 s)</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>User CPU Time (1/60 s)</td>
<td>529</td>
<td>30101</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.000</td>
<td>1.000</td>
<td>1.000</td>
</tr>
<tr>
<td>Iteration</td>
<td>100</td>
<td>500</td>
<td>1.000</td>
</tr>
<tr>
<td>Maximum Error</td>
<td>2.576x10^-4</td>
<td>6.635x10^-4</td>
<td>5.553x10^-4</td>
</tr>
<tr>
<td></td>
<td>System CPU Time (1/60 s)</td>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>User CPU Time (1/60 s)</td>
<td>527</td>
<td>30138</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.981</td>
<td>1.997</td>
<td>1.997</td>
</tr>
<tr>
<td>Iteration</td>
<td>100</td>
<td>500</td>
<td>1.000</td>
</tr>
<tr>
<td>Maximum Error</td>
<td>2.635x10^-4</td>
<td>6.659x10^-4</td>
<td>5.548x10^-4</td>
</tr>
<tr>
<td></td>
<td>System CPU Time (1/60 s)</td>
<td>7</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>User CPU Time (1/60 s)</td>
<td>532</td>
<td>30163</td>
</tr>
<tr>
<td>Speedup</td>
<td>3.933</td>
<td>3.989</td>
<td>3.991</td>
</tr>
<tr>
<td>Iteration</td>
<td>100</td>
<td>500</td>
<td>1.000</td>
</tr>
<tr>
<td>Maximum Error</td>
<td>2.697x10^-4</td>
<td>6.612x10^-4</td>
<td>5.589x10^-4</td>
</tr>
<tr>
<td></td>
<td>System CPU Time (1/60 s)</td>
<td>19</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>User CPU Time (1/60 s)</td>
<td>535</td>
<td>30166</td>
</tr>
<tr>
<td>Speedup</td>
<td>7.653</td>
<td>7.974</td>
<td>7.977</td>
</tr>
<tr>
<td>Iteration</td>
<td>100</td>
<td>500</td>
<td>1.000</td>
</tr>
<tr>
<td>Maximum Error</td>
<td>2.684x10^-4</td>
<td>6.638x10^-4</td>
<td>5.682x10^-4</td>
</tr>
</tbody>
</table>
From Table 1, following observations are made: Although the synchronization time cannot be measured in our simulation, the increase in the system CPU time consumption with the increase in the number of processors indicates the increasing requirement for inter-process communication. The speedup slightly decreases with the increase in the number of processors.

8.7 Closure

We can make the following conclusion from our discussion: 1) the object-oriented programming provides an efficient tool to design and create a new SPARSE_VECTOR class from the existing VECTOR class, 2) using the object of SPARSE_VECTOR class instead of VECTOR class, our proposed concurrent object-oriented solution algorithm saves both CPU time and computer main memory, 3) iterative methods are more effective for concurrent solution of a large system of linear equations mostly due to low interprocess communication.
CHAPTER IX

CONCLUSIONS AND RECOMMENDATIONS

9.1 Summary and Conclusions

The major accomplishments and conclusions drawn from this research can be summarized as follows:

1. A new architecture has been proposed for numerical and finite element analysis. This architecture integrates blackboard architecture, extended entity-relational data model, finite element analysis, version management, and concurrent programming using OOP paradigm.

2. An object-oriented enhanced entity relationship data model has been created for effective processing of a myriad of data types encountered in finite element analysis of complex engineering problems. An object-oriented finite element analysis class library using database management techniques has been developed.

3. A new data management model for complex engineering data objects encountered in FEA has been proposed. This model consists of a data storage structure and a 3+ level index system. The integration of database management system techniques with numerical analysis is particularly useful for managing large quantities of permanent, intermediate, and temporary data.

4. A version management model for numerical analysis in CAE has been proposed. This model is based on a proposed parallel version graph which truly represents
the characteristics of design objects with alternative data in a trial-and-error process.

5. With the basic synchronization techniques and interprocess communication capabilities, a concurrent computing environment is simulated on a UNIX based workstation. An efficient equation solver using object-oriented programming and iterative methods is developed under this environment.

6. The models and concepts developed in this research have been applied to the solution of a complex engineering problem, which is the interlaminar stress analysis of composite laminates.

9.2 Recommendations for Further Research

The application of concurrent object-oriented programming in civil engineering is still in its infancy stage. For further development of this technique, the following recommendations are proposed for further research:

1. Integrating knowledge-based engineering techniques into finite element analysis to develop a smart analysis tool. Trial-and-error results are valuable training examples to neural network learning systems [Adeli and Hung, 1993].

2. Object-oriented programming is an ideal platform for concurrency. With the availability of concurrent object-oriented programming languages, issues related to concurrent programming should be studied. The granularity of process in the level of objects is an example. Another example is object data passing through pipes or message queues. Engineering objects are complex objects. Passing an entire object as one data is easy from the programming point of view. However, not all data in one object are needed. Some information passed is not useful. Establishing a mailing object to deliver necessary information instead of an entire engineering object is one solution.
3. Inputting information for numerical analysis is usually done sequentially by one software system user. With concurrent programming, this procedure can be accomplished by several users as different tasks/ processes. If each task is linked to one separated I/O facility, multiple users can work concurrently to speed-up the entire input procedure.

4. Object-oriented programming provides a useful tool for code reusability. However, many concurrent programs are machine dependent. Taking advantage of inheritance in OOP for COOP needs further study.

5. Lamer [1990] argues: "The parallel execution of knowledge sources is a powerful conceptual principle of blackboard architecture, but it is infrequently present in implementation." This raises the following question: Is it necessary to run all knowledge sources at one time, or wait and select better hypothesis from completed execution for further computation? Applying this to our design process, it is not necessary to run all possible versions of design objects in parallel. We can take the general steps of making a few trials, comparing them, and then trying more versions. Integrating heuristics in the parallelism organization is an interesting topic.

6. The convergence of the blackboard architecture needs further study. Blackboard problem solving procedure is based on hypotheses. The correctness of these hypotheses determines the convergence of the solution.

7. The blackboard architecture encourages concurrency. However, the granularity of parallelism is problem and hardware dependent. The granularity of knowledge sources should be studied.
LIST OF REFERENCES


Coad, P. and Yourdon, E. (1990), Object-Oriented Analysis, Prentice-Hall, Englewood Cliffs, NJ.


Ezzell, B. (1989), Object-Oriented Programming in Turbo Pascal 5.5, Addison-Wesley, Reading, MA.


Keffer, T. (1991), "Why C++ will replace FORTRAN (or, at least, why it should)". Rogue Wave Software, Corvallis, OR.


Norrie, M. (1993?), " Supporting the Design of Object-Oriented Database".


Wilkes, W., Klahold, P., and Schlageter, G. (1989), "Complex and Composite Objects in CAD/CAM Databases", Proc. of Fifth International Conf. on Data Engineering, Feb. 6-10, Los Angeles, CA.


APPENDIX A
CLASS LIBRARY OF VECTOR COMPUTATION

A.1 Source Code of VECTOR.H

#include "general.h"

#define ROW 0  
#define COLUMN 1

class VECTOR{
    public:
    // Constructors and destructor
    VECTOR();
    VECTOR(int);
    VECTOR(int, double ...);
    VECTOR(int, double*);
    VECTOR(const VECTOR&);
    VECTOR(const VECTOR&, int);
    -VECTOR();

    // Methods
    VECTOR& setUp(void);
    VECTOR& SetUp(int);
VECTOR& MakeZero();
VECTOR& KeyIn(char*);
VECTOR& MakeUnit(int);
VECTOR& AllEntries(const double&);
VECTOR& Copy(const VECTOR&, int);
VECTOR& FtoV(const double*, int);

void AssignShmID(int, int, int);
void DetachShm(void);
void VtoF(double*, int);
void PrintForm(int k) {form = k;};
void AssignName(char* s) {strcpy(Name,s);};
void AssignFile_name(char* s) {strcpy(File_name,s);};
Bool Store(char*, streampos); // position of vector head
Bool Retrieve(char*, streampos); // position of vector head
Bool Delete(char*, streampos); // Entire vector
streampos Insert(char*, streampos); // Entire vector
 Bool Update(char*, streampos, int, double);
double Retrieve(char*, streampos, int);

int DimIndex() const {return Dim;};
int get_ShmArray() const {return ShmArray;};
int get_ShmName() const {return ShmName;};
int get_ShmFile_name() const {return ShmFile_name;};

int get_form() const {return form;};
char* get_Name(void) const {return Name;};
char* get_File_name(void) const {return File_name;};
streampos get_Length(void) const {return Length;};
streampos get_Offset(void) const {return Offset;};
Bool get_ACTIVE(void) const {return ACTIVE;};

double get_max(void);
double get_min(void);

// Overloaded Operators
double& operator [] (int)const;
VECTOR& operator +=(const VECTOR&);
VECTOR& operator -==(const VECTOR&);
VECTOR& operator *==(const double&);
VECTOR& operator =(const VECTOR&);

friend ostream& operator<<(ostream&, const VECTOR&);
friend int operator==(const VECTOR&, const VECTOR&);
friend int operator!=(const VECTOR&, const VECTOR&);
friend VECTOR operator *(const double&, const VECTOR&);
friend VECTOR operator +(const VECTOR&, const VECTOR&);
friend double operator *(const VECTOR&, const VECTOR&);

protected:
// storage information
A.2 Source Code of VECTOR.C

 ifndef VECTOR_C
 define VECTOR_C

 /**********************************************************
 Definition of class VECTOR

 File name: vector.c 9/30/93

 Coded by George G. Yu (The Ohio State University)

 Compiled by «AT&T C++ Translator 2.1.03 08/31/90»

 Ref:
 1) James T. Smith (C++ for Scientists and Engineers,
 2) Barry Jaspan, MIT, bjaspan@mit.edu

 **********************************************************/

 #include "vector.h"

 // Constructor and destructor
 /
 VECTOR::VECTOR()
 {
  Dim= -1;
  Array= NULL;
}
VECTOR::VECTOR(int Hi)
{
   Share= NO;
   SetUp(Hi);
   if (GetError()) return;
}

VECTOR::VECTOR(int Hi, double first ...)
{
   int i;
   va_list ap;
   Share= NO;
   SetUp(Hi);
   if (GetError()) return;

   Array[0]= first;
   va_start(ap,first);
   for (i= 1; i<Dim; i++)
      Array[i]= double(va_arg(ap,double));
}

VECTOR::VECTOR(int Hi, double *array)
{
   Share= NO;
   SetUp(Hi);
   if (GetError()) return;
   memcpy(Array,array,Dim*sizeof(double));
}

VECTOR::VECTOR(const VECTOR& Source)
{
   Share= NO;
   SetUp(Source.DimIndex());
   if (GetError()) return;
   memcpy(Array,Source.Array,Dim*sizeof(double));
}

VECTOR::VECTOR(const VECTOR& Source, int NewDim)
{
   int i;
   Share= NO;
SetUp(NewDim);
if (GetError()) return;
if (Dim==NewDim)
{
    memcpy(Array, Source.Array, Dim*sizeof(double));
}
else if (Dim > NewDim)
{
    for (i=0; i<NewDim; i++)
    Array[i]= Source[i];
}
else
    for (i=0; i<Dim; i++)
    Array[i]= Source[i];

VECTOR::~VECTOR()
{
    // cout<<"inside vector’s destructor"<<endl;
    if (Array!= NULL && Share== NO)
        delete[] Array;
    else if (Share== YES)
        shmdt((char*)Array);
    if (File_name != NULL && Share== NO)
        delete[] File_name;
    else if (File_name != NULL && Share== YES)
        shmdt((char*)File_name);
    if (Name != NULL && Share== NO)
        delete[] Name;
    else if (Name != NULL && Share== YES)
        shmdt((char*)Name);
}

// Methods
VECTOR& VECTOR::SetUp(void)
{
    SetUp(Dim);
    return This;
}
VECTOR& VECTOR::SetUp(int Hi)
{
    Dim= Hi;
if (Share == NO)
{
    Array = new double[Dim];
    File_name = new char[15];
    Name = new char[15];
    if (Dim > 0 && Array == NULL)
    {
        SetError(OutOfMemory);
        Dim = 0;
    }
    for (int i = 0; i < Dim; i++)
        Array[i] = 0.;
}
else
{
    if (ShmArray != 0)
        Array = (double*) shmat(ShmArray, 0, 0);
    if (ShmFile_name != 0)
        File_name = (char*) shmat(ShmFile_name, 0, 0);
    if (ShmName != 0)
        Name = (char*) shmat(ShmName, 0, 0);
}
form = ROW;
Type = OVECTOR;
return This;

VECTOR& VECTOR::MakeZero()
{
    AllEntries(0.0);
    return This;
}

VECTOR& VECTOR::KeyIn(char* Name)
{
    for (int i = 0; i < Dim; i++)
    {
        cout << Name << '[' << i << ']':
        cin >> This[i];
    }
    return This;
}

VECTOR& VECTOR::MakeUnit(int k)
```cpp
{
  if(0 <= k && k < Dim)
  {
    MakeZero();
    This[k] = 1.0;
  }
  else
    setError(IndexError);
  return This;
}

 Douglas W. O'Leary & Jerry B. Rawlinson

VECTOR & VECTOR::AllEntries(const double & x)
{
  for (int i = 0; i < Dim; i++)
    This[i] = x;
  return This;
}

VECTOR & VECTOR::Copy(const VECTOR & Source, int SourceDim)
{
  memcpy(&This[0], &Source[0], SourceDim*sizeof(double));
  return This;
}

void VECTOR::AssignShmID(int ArrayID, int NameID, int FileID)
{
  Share = YES;
  ShmArray = ArrayID;
  ShmName = NameID;
  ShmFile_name = FileID;
}

void VECTOR::DetachShm(void)
{
  //cout << "inside VECTOR::DetachShm " << Name << endl;
  if(Array != NULL)
    if(shmdt((char*)Array) < 0)
      perror("Detach vector Array error");
  if(ShmName != 0)
    if(shmdt((char*)Name) < 0)
      perror("Detach vector Name error");
  if(ShmFile_name != 0)
    if(shmdt((char*)File_name) < 0)
      perror("Detach vector File_name error");
}
VECTOR& VECTOR::FtoV(const double* A, int m)
{
    int i;
    for(i= 0; i< m; i++)
        This[i]= *(A+i);

    return This;
}

void VECTOR::VtoF(double* A, int m)
{
    int i;
    for(i= 0; i< m; i++)
        *(A+i)= This[i];
}

double VECTOR::get_max(void)
{
    int i;
    double max;
    max= This[0];

    for(i=1; i<Dim; i++)
        if(This[i]>max)
            max= Array[i];

    return max;
}

double VECTOR::get_min(void)
{
    int i;
    double min;
    min= This[0];

    for(i=1; i<Dim; i++)
        if(This[i] < min)
            min= Array[i];

    return min;
}

Bool VECTOR::Store(char* f_name, streampos Start)
{
int i;
int size_d = sizeof(double);
int size_V = sizeof(VECTOR);
fstream f;

This.ACTIVE = True;
strcpy(File_name, f_name);
f.open(File_name, ios::in | ios::out);
if(!f)
{
    cout<<"Cannot open file " << f_name << endl;
    return False;
}
else
{
    f.seekp(Start, ios::beg);
    f.write((char**)&(This), size_V);
    Offset = Start + size_V;

    f.seekp((Start + size_V), ios::beg);
    for(i=0; i<Dim; i++)
    {
        f.write((char*)(&(This[i]), size_d));
    }

    Length = f.tellp() - Offset;
    f.seekp(Start, ios::beg);
    f.write((char*)(&(This).size_V));

    f.close();
    return True;
}

Bool VECTOR::Retrieve(char* f_name, streampos Start)
{
    int i;
    int size_d = sizeof(double);
    int size_V = sizeof(VECTOR);
    fstream f;

    f.open(f_name, ios::in | ios::out);
    if(!f)
    {
        cout<<"Cannot open file " << f_name << endl;
        return False;
    }
    else
{  
  f.seekg(Start, ios::beg);  
  f.read((char*)&(This), size_V);  

  if(This.ACTIVE== True)  
  {  
    This.SetUp();  

    f.seekg(Start+streampos(size_V), ios::beg);  
    for(i=0; i<Dim; i++)  
      f.read((char*)&(This[i]).size_d);  

    if((f.tellg()-Offset)!= Length)  
    {  
      cout<<"The record is wrong."<<endl;  
    }  
    f.close();  
    return True;  
  }  
  else  
  {  
    return False;  
  }  
}  

 ////////////////////////////////////////////  
 Bool VECTOR::Delete(char* f_name, streampos Start)  
 {  
  int size_V= sizeof(VECTOR);  
  fstream f;  

  f.open(f_name,ios::in|ios::out);  
  if(!f)  
    cout<<"Cannot open file ">>f_name<<endl;  

  f.seekg(Start, ios::beg);  
  f.read((char*)&(This), size_V);  

  if(This.ACTIVE== True)  
  {  
    This.ACTIVE= False;  
    f.seekp(Start, ios::beg);  
    f.write((char*)&(This), size_V);  
    f.close();  
    return True;  
  }  
  else  
  {  
    f.close();  
  }  
}
return False;
}
}

streampos VECTOR::Insert(char* f_name, streampos Start)
{
    int size_V = sizeof(VECTOR);
    fstream f;
    VECTOR temp;
    streampos New_Start, End;

    // Check the space available
    f.open(f_name, ios::in|ios::out);
    if(!f)
    {
        cout<<"WARNING: Cannot open file ":<<f_name<<endl;
        return -1;
    }
    else
    {
        f.seekg(0, ios::end);
        End = f.tellg();
        if(Start!=-1)
        {
            if(End> Start) // avoiding out of file
            {
                f.seekg(Start, ios::beg);
                f.read((char*)&temp, size_V);
                if(temp.getLength() < (This.DimIndex()*sizeof(double)))
                    New_Start = End;
                else
                    New_Start = Start;
            }
            else
                New_Start = Start;
        }
        else
            New_Start = End;
        f.close();

        // Store the VECTOR
        if(This.Store(f_name, New_Start) == TRUE)
            return New_Start;
        else
            return -1;
    }
}
bool VECTOR::Update(char* f_name, streampos Start, int C, double d)
{
    int size_V = sizeof(VECTOR);
    fstream f;

    f.open(f_name, ios::in | ios::out);
    if(!f)
    {
        cout<<"Cannot open file "<<f_name<<endl;
        return false;
    }
    else
    {
        f.seekg(Start, ios::beg);
        f.read((char*)&This, size_V);

        if(This.ACTIVE == true)
        {
            streampos temp;
            temp = This.Get_Offset() + C * sizeof(double);

            f.seekp(temp, ios::beg);
            f.write((char*)&d, sizeof(double));
            f.close();
            return true;
        }
        else
        {
            return false;
        }
    }
}

double VECTOR::Retrieve(char* f_name, streampos Start, int C)
{
    int size_V = sizeof(VECTOR);
    fstream f;
    double data;

    f.open(f_name, ios::in);
    if(!f)
    {
        cout<<"Cannot open file "<<f_name<<endl;
        return 0;
    }

    f.seekg(Start, ios::beg);
    f.read((char*)&This, size_V);
    return This.GetData();
f.read((char*)&(This), size_V);

if(This.ACTIVE== TRUE)
{
    streampos temp;
    temp= This.get_Offset()+C*sizeof(double);

    f.seekg(temp, iosv.beg);
    f.read((char*)&data, sizeof(double));
    f.close();
    return data;
}
else
{
    cout<<" WARNING: The VECTOR does not exist!"<<endl;
    return 0;
}

// Overloaded Operators

// Overloaded Operators
double& VECTOR::operator [] (int k) const
{
    if(k< 0 || k>= Dim)
        cout<< "Index ">= Dim"");
//cout<<"inside operator [] k="<<k<<endl;
//cout<<"Array[] = "<<*(Array+k)<<endl;
    return *(Array+k);
}

VECTOR& VECTOR::operator+=(const VECTOR& Source)
{
    int i;

    if(Source.DimIndex()==-1)
    {
        cout<< "Add a NULL vector" << endl;
    }
    else if (Source.DimIndex()!= Dim)
    {
        cout<< "Two vectors have different dimensions"<< endl;
    }
    else
    for (i= 0; i< Dim; i++)
        Array[i] += Source[i];

    return This;
VECTOR & VECTOR::operator=(const VECTOR & Source)
{
    int i;

    if(Source.DimIndex()==-1)
    {
        cout << "A NULL source vector" << endl;
    }
    else if(Source.DimIndex()!=Dim)
    {
        cout << "Two vectors have different dimensions" << endl;
    }
    else
    {
        for (i=0; i<Dim; i++)
            Array[i] = Source[i];

        return This;
    }
}

VECTOR & VECTOR::operator*=(const double & x)
{
    int i;

    for (i=0; i<Dim; i++)
        Array[i] *= x;

    return This;
}

VECTOR & VECTOR::operator-=(const VECTOR & Source)
{
    int i;

    if(Source.DimIndex()==-1)
    {
        cout << "Add a NULL vector" << endl;
    }
    else if(Source.DimIndex()!=Dim)
    {
        cout << "Two vectors have different dimensions" << endl;
    }
    else
    {
        for (i=0; i<Dim; i++)
            Array[i] -= Source[i];

        return This;
    }
}
return This;
}

ofstream & operator<<(ostream & s, const VECTOR & Source)
{
    int i;
    int Hi = Source.dimIndex();

    if(Hi == -1)
        s << "The vector is NULL."
    else
    {
        s << "Vector[0,""HI-1""] is:
        if(Source.get_form() == ROW)
            for (i = 0; i < Hi; i++)
                s << Source[i] << " ";
        s << endl << endl;
        else
        {
            for (i = 0; i < Hi; i++)
                s << Source[i] << endl;
            s << endl;
        }
        return(s);
    }

    // Related non-member logic function

    int operator==(const VECTOR & s1, const VECTOR & s2)
    {
        if(s1.dimIndex() == s2.dimIndex())
            for (int i = 0; i <= s1.dimIndex(); i++)
                if(s1[i] != s2[i])
                    return FALSE;
        else
            return FALSE;
        return TRUE;
    }

    int operator!=(const VECTOR & s1, const VECTOR & s2)
if(s1.DimIndex() != s2.DimIndex())
{
    for (int i = 0; i <= s1.DimIndex(); i++)
        if(s1[i] == s2[i])
            return FALSE;
    else
        return FALSE;
return TRUE;
}

VECTOR operator *(const double& c, const VECTOR & W)
{
    VECTOR Sum(W.DimIndex());
    for (int i = 0; i < W.DimIndex(); i++)
        Sum[i] = c*W[i];
    return Sum;
}

VECTOR operator +(const VECTOR & V, const VECTOR & W)
{
    VECTOR Sum(V.DimIndex());
    for (int i = 0; i < V.DimIndex(); i++)
        Sum[i] = V[i] + W[i];
    return Sum;
}

double operator *(const VECTOR & V, const VECTOR & W)
{
    double temp = 0.0;
    if(V.DimIndex() != W.DimIndex())
        cout << "Two vectors have different dimensions." << endl;
    else
    {
        for (int i = 0; i < V.DimIndex(); i++)
            temp += V[i] * W[i];
    }
    return temp;
}
A.3 Source Code of SPARSE_VECTOR.H

#ifndef SPARSE_MATRIX_H
#define SPARSE_MATRIX_H

// Definition of class SPARSE_MATRIX

File name: smatrix.h 1/2/93
Coded by George G. Yu (The Ohio State University)
Compiled by IBM C++ compiler XLC

#include "general.h"
#include "svector.h"
#include "matrix.h"

class SPARSE_MATRIX : public MATRIX
{
public:
    SPARSE_MATRIX(int, int);
    SPARSE_MATRIX(void);
    ~SPARSE_MATRIX(void);
    SPARSE_MATRIX& SetUp(int, int);
    SPARSE_MATRIX& SetUp(int, int, int, short*);

    // overloaded operator
    SPARSE_VECTOR& operator[](int) const;

    friend ostream& operator<<(ostream&, const SPARSE_MATRIX&);

protected:
    SPARSE_VECTOR *Row;
};

#endif

A.4 Source Code of SPARSE_VECTOR.C

/*************************************************************/
Definition of class SPARSE_VECTOR

File Name: svector.C 1/2/93

Coded by George G. Yu (The Ohio State University)

Compiled by IBM Risc/6000 C++ compiler xIC

#include "svector.h"

// Constructors

SPARSE_VECTOR::SPARSE_VECTOR(void) : VECTOR()
{
    Fill_in_Index=NULL;
}

SPARSE_VECTOR::SPARSE_VECTOR(int Dim, int Fill_Dim,
    short *Fill_in)
    : VECTOR(Fill_Dim)
{
    SetUp(Fill_in);
    Size= Dim;
    if(GetError()) return;
}

SPARSE_VECTOR::SPARSE_VECTOR(const SPARSE_VECTOR& Source)
    : VECTOR(Source.DimIndex())
{
    SetUp(Source.get_Fill_in_Index());
    if(GetError()) return;
    Size= Source.Size;
    memcpy(Array, Source.Array, Dim*sizeof(double));
    memcpy(Fill_in_Index, Source.Fill_in_Index,
        Dim*sizeof(short));
}

SPARSE_VECTOR::~SPARSE_VECTOR(void)
{ 
  if (Fill_in_Index != NULL) 
    delete Fill_in_Index;
}

// SPARSE_VECTOR & SPARSE_VECTOR::SetUp(short * Fill_in)
// SPARSE_VECTOR & SPARSE_VECTOR::SetUp(short * Fill_in)
{
  int i;

  Fill_in_Index = new short[Dim];
  Type = OSVECTOR;

  for (i=0; i<Dim; i++)
    Fill_in_Index[i] = Fill_in[i];

  VECTOR::SetUp(Dim);
  return This;
}

// SPARSE_VECTOR & SPARSE_VECTOR::SetUp(int dim, int fill, 
// short * Fill_in)
{
  int i;

  Dim = fill;
  Size = dim;
  Fill_in_Index = new short[Dim];
  Type = OSVECTOR;

  for (i=0; i<Dim; i++)
    Fill_in_Index[i] = Fill_in[i];
  VECTOR::SetUp(Dim);

  return This;
}

// double& SPARSE_VECTOR::operator [](int k) const
{
  int i;
  double aa = 0.0;
  double *a = &aa;

  for (i=0; i<Dim; i++)
  {
if(k == Fill_in_Index[i])
    a = Array+i;
}

return *a;

SPARSE_VECTOR& SPARSE_VECTOR::operator += (const SPARSE_VECTOR& Source)
{
    int i;
    if(Source.Size == Size && Source.Dim == Dim)
    {
        for (i=0; i<Dim; i++)
            if (Source.Fill_in_Index[i] != Fill_in_Index[i])
            {
                cout<<"Warning: Two sparse matrices are not compatible!";
                cout<<" No addition is made."<<endl;
                return This;
            }
        for (i=0; i<Dim; i++)
            Array[i] += Source.Array[i]; // not Source[i], it is a full vector
    }
    return This;
}

SPARSE_VECTOR& SPARSE_VECTOR::operator -= (const SPARSE_VECTOR& Source)
{
    int i;
    if(Source.Size == Size && Source.Dim == Dim)
    {
        for (i=0; i<Dim; i++)
            if (Source.Fill_in_Index[i] != Fill_in_Index[i])
            {
                cout<<"Warning: Two sparse matrices are not compatible!";
                cout<<" No subtraction is made."<<endl;
                return This;
            }
        for (i=0; i<Dim; i++)
            Array[i] -= Source.Array[i]; // not Source[i], it is a full vector
    }
}
SPARSEVECTOR& SPARSEVECTOR::operator *= (const double& x)
{
    int i;
    for (i=0; i<Dim; i++)
        Array[i] *= x;
    return This;
}

ostream& operator << (ostream& s, const SPARSEVECTOR& Source)
{
    int i;
    s<"This is a sparse vector with "<Source.DimIndex()<" fill in's out of "<Source.Size<" entries.";

    if(Source.get_form() == ROW)
        for (i=0; i< Source.Size; i++)
            s<Source[i]<", ";
        s<endl<<endl;
    else
        for (i= 0; i< Source.Size; i++)
            s< Source[i]<<endl;
        s<endl;

    return (s);
}

double operator * (const SPARSEVECTOR& V, const VECTOR& W)
{
    double temp= 0.0;
    int i, k;
if(V.Size != W.DimIndex())
{
    cout<<"Warning: Two vectors have different dimensions."
    cout<<V:"" V.Size"" W:"" W.DimIndex()"<endl;
}
else
    for (i=0; i<V.DimIndex(); i++)
    {
        k= V.Fill_in_Index[i];
        temp += V.Array[i] * W[k];
    }
 return temp;

double operator *(const SPARSE_VECTOR& V,
        const SPARSE_VECTOR& W)
{
    double temp= 0.0;
    int i, kv, kw;
    if(V.Size!=W.Size)
        cout<<"Warning: Two vectors have different dimensions."
    else
        {
            // This routine assumes two vectors having different fill in.
            int *m= new int[V.Size];
            for(i=0; i<V.Size; i++)
                m[i]= 0;
            // Marking the fill in entry with 1.
            for(i=0; i<V.DimIndex(); i++)
                m[V.Ffill_in_Index[i]]= 1;
            //
            for (i=0; i<W.DimIndex(); i++)
            {
                if(m[W.Fill_in_Index[i]]== 1)
                {
                    cout<<"W.Fill_in_Index[i]= "<W.Fill_in_Index[i]<<endl;
                    temp += V.Array[i] * W.Array[i];
                    cout<<"V= "V.Array[i] " W= "W.Array[i]<<endl;
                }
            }
            delete m;
        }
return temp;
}

vector operator + (const sparse_vector& V,  
const vector& W)
vector Sum(W);
for (i=0; i<V.dim_index(); i++)
  Sum[V.fill_in_index[i]] += V.array[i];
return Sum;
}

vector operator - (const sparse_vector& V,  
const vector& W)
vector Sum(W);
for (i=0; i<V.dim_index(); i++)
  Sum[V.fill_in_index[i]] -= V.array[i];
return Sum;
}

double operator * (const vector& W,  
const sparse_vector& V)
double temp= 0.0;
int i, k;
if(V.size != W.dim_index())
cout<<"Warning: Two vectors have different dimensions."<<endl;
else
for (i=0; i<V.DimIndex(); i++)
{
   k= V.Fill_in_Index[i];
   temp += V.Array[i] * W[k];
}
return temp;


VECTOR operator + (const VECTOR& W,
 const SPARSE_VECTOR& V)

{ int i;
  VECTOR Sum(W);
  for (i=0; i<V.DimIndex(); i++)
    Sum[V.Fill_in_Index[i]] += V.Array[i];
  return Sum;
}

VECTOR operator - (const VECTOR& W,
 const SPARSE_VECTOR& V)

{ int i;
  VECTOR Sum(W);
  for (i=0; i<V.DimIndex(); i++)
    Sum[V.Fill_in_Index[i]] -= V.Array[i];
  return Sum;
}
APPENDIX B
CLASS LIBRARY OF MATRIX COMPUTATION

B.1 Source Code of MATRIX.H

#ifndef MATRIX_H
#define MATRIX_H

#define MATRIX_H

#include "general.h"
#include "vector.h"

class MATRIX
{
public:

// Constructors and destructor
MATRIX();
MATRIX(int, int);
MATRIX(int, int, char*);
MATRIX(const MATRIX&);
MATRIX(int, int, double*);
-MATRIX(const MATRIX&, int, int, int);
-MATRIX();

// Methods
MATRIX& MakeZero(void);

};

Ref:
1) James T. Smith (C++ for Scientists and Engineers,
2) Barry Jaspan, MIT, bjaspan@mit.edu
MATRIX& MakeUnit(void);
MATRIX& KeyIn(char*);
MATRIX& SetUp(int, int);
MATRIX& SetUp(void);
MATRIX& Copy(const MATRIX&);
MATRIX& Add(const MATRIX&, int, int);
MATRIX& Sub(const MATRIX&, int, int);
MATRIX& FtoM(const double*, int, int);

void MtoF(double*, int, int);
Bool Store(char*, streampos);  // position of matrix head
Bool Retrieve(char*, streampos);  // position of matrix head
Bool Delete(char*, streampos);  // Entire matrix
Bool Update(char*, streampos, int, int, double);
streampos Insert(char*, streampos&);  // Entire matrix
double Retrieve(char*, streampos, int, int);
void set_form(int i) {form=i;};
void assign_Name(char* s) {strcpy(Name,s);};
void assign_File_name(char* s) {strcpy(File_name,s);};

int CdimIndex(void) const {return Cdim;};
int RdimIndex(void) const {return Rdim;};
int get_form(void) const {return form;};
char* get_Name(void) const {return Name;};
char* get_File_name(void) const {return File_name;};
streampos get_Length(void) const {return Length;};
streampos get_Offset(void) const {return Offset;};
Bool get_ACTIVE(void) const {return ACTIVE;};

// Overloaded operator
VECTOR& operator [](int) const;
MATRIX& operator =(const MATRIX&);
MATRIX& operator +=(const MATRIX&);
MATRIX& operator -=(const MATRIX&);
MATRIX& operator *(const double&);
friend MATRIX operator +(const MATRIX&, const MATRIX&); // [a]+[b] = [a+b]
friend MATRIX operator -(const MATRIX&, const MATRIX&); // [a]-[b] = [a-b]
friend MATRIX operator *(const MATRIX&, const MATRIX&); // [a]*[b] = [a*b]
friend MATRIX operator *(const double, const MATRIX&);
friend VECTOR operator *(const VECTOR&, const MATRIX&);
friend VECTOR operator *(const MATRIX&, const VECTOR&);
friend MATRIX& operator*=(MATRIX&, const MATRIX&); // [a]*[b] = [a*b]
friend MATRIX operator [[const MATRIX&, const MATRIX&]](const MATRIX&); // [a][b] = [a][b]
friend MATRIX operator &=(const MATRIX&, const MATRIX&);
friend int operator ==(const MATRIX&, const MATRIX&);
friend ostream& operator <<(ostream&, const MATRIX&);
protected:
// storage information
Object_type Type;
Bool ACTIVE;
streampos Offset; // position of This[0][0]
streampos Length;
char *File_name;

int Rdim;
int Cdim;
int form;
VECTOR *Row;
char *Name;

};

MATRIX inverse(const MATRIX&);
MATRIX trans (const MATRIX&);

Ref:
1) James T. Smith (C++ for Scientists and Engineers.
2) Barry Jaspan, MIT. bjaspan@mit.edu

#include "matrix.h"

MATRIX::MATRIX()
{  
  Rdim= 0;
  Cdim= 0;
  Row= NULL;
  Name= NULL;
  File_name= NULL;
}

MATRIX::MATRIX(int R, int C)
{  
  Name= NULL;
  File_name= NULL;
  SetUp(R,C);
  if (GetError()) return;
}

MATRIX::MATRIX(int R, int C, char *name)
{  
  File_name= NULL;
  SetUp(R,C);
  assign_Name(name);
  if (GetError()) return;
}

MATRIX::MATRIX(const MATRIX& Source)
{  
  SetUp(Source.RdimIndex(), Source.CdimIndex());
  if (GetError()) return;
  Copy(Source);
}

MATRIX::MATRIX(int RowHi, int ColHi, double *array)
```c
/**************/
{
    int i, j, n;

    Name = NULL;
    File_name = NULL;
    SetUp(RowHi, ColHi);
    if (GetError()) return;
    n = 0;
    for (i = 0; i < RowHi; i++)
        for (j = 0; j < ColHi; j++)
            This[i][j] = array[n];
    n++;
}

MATRIX::MATRIX(const MATRIX& Source, int NewRow, int NewCol, int OldRow, int OldCol)
/**************/
{
    int i, j;

    Name = NULL;
    File_name = NULL;
    SetUp(NewRow, NewCol);
    if (GetError()) return;

    for (i = 0; i < NewRow; i++)
        for (j = 0; j < NewCol; j++)
            This[i][j] = Source[i+OldRow][j+OldCol];
}

MATRIX::~MATRIX()
/**************/
{
    if (Row != NULL)
        delete[] Row;
    if (File_name != NULL)
        delete File_name;
    if (Name != NULL)
        delete Name;
}

// Methods
/**************/
MATRIX& MATRIX::MakeZero()
/**************/
```
{ 
    AllEntries(0.0); 
    return This; 
}

COPYRIGHT

MATRIX& MATRIX::MakeUnit()
{ 
    int i;
    if(Rdim==Cdim)
    { 
        MakeZero();
        for(i = 0; i < Rdim; i++)
            This[i][i] = 1.0;
    }
    else
        cout<<"Rdim!= Cdim. No Unified matrix."<<endl;

    return This;
}

MATRIX& MATRIX::KeyIn(char* Name)
{  
    int j;
    for (int i = 0; i < Rdim; i++)
        for (j = 0; j < Cdim; j++)
        { 
            cout<< Name <<"["<<i<<"]"<<j<<"]: ";
            cin>> This[i][j];
        }

    return This;
}

MATRIX& MATRIX::SetUp(int RowHi, int ColHi)
{ 
    Rdim= RowHi;
    Cdim= ColHi;

    File_name= new char[15]; 
    Name= new char[15];

    Row= new VECTOR[Rdim];
    if (RowHi>0 && Row==NULL)
    { 
        SetError(OutOfRange);
        Rdim= 0;
    }
for (int i = 0; i < RowHi; i++)
    This[i].SetUp(ColHi);

Type = OMATRIX;
return This;

/**
 * MATRIX& MATRIX::SetUp(void)
 */
MATRIX& MATRIX::SetUp(void)
{
    SetUp(Cdim, Rdim);
    return This;
}

/**
 * MATRIX& MATRIX::Copy(const MATRIX& Source)
 */
MATRIX& MATRIX::Copy(const MATRIX& Source)
{
    int i, j;
    Rdim = Source.RdimIndex();
    Cdim = Source.CdimIndex();

    for (i = 0; i < Rdim; i++)
        for (j = 0; j < Cdim; j++)
            This[i][j] = Source[i][j];
    strcpy(This.Name, Source.get_Name());
    strcpy(This.File_name, Source.get_File_name());
    return This;
}

/**
 * MATRIX& MATRIX::AllEntries(const double& x)
 */
MATRIX& MATRIX::AllEntries(const double& x)
{
    int i, j;
    for (i = 0; i < Rdim; i++)
        for (j = 0; j < Cdim; j++)
            This[i][j] = x;
    return This;
}

/**
 * MATRIX& MATRIX::Add(const MATRIX& V, int ir, int ic)
 */
MATRIX& MATRIX::Add(const MATRIX& V, int ir, int ic)
{
    int i, j;
if(V.RdimIndex()+ir > Rdim)
{  
    cout<< "Two matrices have different ROW dimension!!\n\n"<< endl;
    exit(-1);
}
else if(V.CdimIndex()+ic > Cdim)
{  
    cout<< "Two matrices have different COLUMN dimension!!\n\n"<< endl;
    exit(-1);
}
else
{
    for (i= 0; i< V.RdimIndex(); i++)
        for (j= 0; j< V.CdimIndex(); j++)
            This[i+ir][j+ic] += V[i][j];
}
return This;

/**
MATRIX& MATRIX::Sub(const MATRIX& V, int ir, int ic)
/**
{
    int i, j;

    if(V.RdimIndex()+ir > Rdim)
    {  
        cout<< "Two matrices have different ROW dimension!!\n\n"<< endl;
        exit(-1);
    }
    else if(V.CdimIndex()+ic > Cdim)
    {  
        cout<< "Two matrices have different COLUMN dimension!!\n\n"<< endl;
        exit(-1);
    }
    else
    {
        for (i= 0; i< V.RdimIndex(); i++)
            for (j= 0; j< V.CdimIndex(); j++)
                This[i+ir][j+ic] -= V[i][j];
    }
    return This;
}
/**
MATRIX& MATRIX::FroM(const double* A, int m, int n)
/**
{
    int i,j;
for(i=0; i<m; i++)
    for(j=0; j<n; j++)
        This[i][j] = *(A+i+j*m);

return This;

/*******************************************************************************/
void MATRIX::MtoF(double* A, int m, int n)
/*******************************************************************************/
{
    int i,j;
    for(i=0; i<m; i++)
        for(j=0; j<n; j++)
            *(A+i+j*m) = This[i][j];
}

/*******************************************************************************/
Bool MATRIX::Store(char* f_name, streampos Start)
/*******************************************************************************/
{
    int i, j;
    int size_d = sizeof(double);
    int size_M = sizeof(MATRIX);
    fstream f;

    This.ACTIVE = True;
    strcpy(File_name, f_name);
    f.open(File_name, ios::in|ios::out);
    if(!f)
        {cout<<"Cannot open file "<<f_name<<endl;
         return False;
        }

    Offset = Start+size_M;
    f.seekp((Start+size_M), ios::beg);

    for(i=0; i<Rdim; i++)
        for(j=0; j<Cdim; j++)
            f.write((char*)&(This[i][j]), size_d);

    Length = f.tellp()-Offset;
    f.seekp(Start, ios::beg);
    f.write((char*)&(This), size_M);

    f.close();
    return True;
Bool MATRIX::Retrieve(char* f_name, streampos Start)
{
    int i, j;
    int size_d= sizeof(double);
    int size_M= sizeof(MATRIX);
    fstream f;

    f.open(f_name,ios::in);
    if(!f)
    {
        cout<<"Cannot open file "<<f_name<<endl;
        return False;
    }

    f.seekg(Start, ios::beg);
    f.read((char*)&(This), size_M);

    if(This.ACTIVE== True)
    {
        This.SetUp();

        for(i=0; i<Rdim; i++)
            for (j=0; j<Cdim; j++)
                f.read((char*)&(This[i][j]), size_d);

        if((f.tellg()-Offset)!= Length)
        {
            cout<<"The record is wrong."<<endl;
        }
        f.close();
        return True;
    }
    else
    return False;
}

Bool MATRIX::Delete(char* f_name, streampos Start)
{
    int size_M= sizeof(MATRIX);
    fstream f;

    f.open(f_name,ios::in|ios::out);
cout<<"Cannot open file "<<f_name<<endl;

f.seekg(Start, ios::beg);
f.read((char*)&(This), size_M);

if(This.ACTIVE== True)
{
    This.ACTIVE= False;
    f.seekp(0, ios::beg);
    f.write((char*)&(This), size_M);
    f.close();

    return True;
}
else
{
    f.close();
    return False;
}
(This.CdimIndex() * This.RdimIndex() * sizeof(double))
New_Start = End;
else
{
    if(temp.get_ACTIVE() == YES)
    {
        cout << "Are you sure to override the existing record?"
            << "(0/1) (NO/YES)" << endl;
        cin >> input;
        if(input == NO)
            New_Start = End;
        else
            New_Start = Start;
    }
    else
        New_Start = Start;
}
else
    New_Start = Start;
else
    New_Start = End;
f.close();

// Store the matrix
if(This.Store(f_name, New_Start) == TRUE)
    return New_Start;
else
    return -1;

/*********************/
Bool MATRIX::Update(char* f_name, streampos Start,
    int C, int R, double d)
/*********************/
{
    int size_M = sizeof(MATRIX);
    fstream f;

    f.open(f_name, ios::in|ios::out);
    if(!f)
    {
        cout << "Cannot open file " << f_name << endl;
        return False;
    }

    f.seekg(Start, ios::beg);
    f.read((char*)&(This), size_M);
}
if(This.ACTIVE== True)
{
    streampos temp;
    temp= This.get_Offset()+(R*This.CdimIndex()+C)*sizeof(double);
    
    f.seekp(temp, ios::beg);
    f.write((char*)&d, sizeof(double));
    f.close();
    return True;
}
else
    return False;
}

double MATRIX::Retrieve(char^ f_name, streampos Start, int C, int R)
{
    int size_M= sizeof(MATRIX);
    fstream f;
    double data;
    
    f.open(f_name, ios::in);
    if(!f)
    {
        cout<<"Cannot open file "<f_name<<endl;
        return 0;
    }
    
    f.seekg(Start, ios::beg);
    f.read((char*)&(This), size_M);
    if(This.ACTIVE== TRUE)
    {
        streampos temp;
        temp= This.get_Offset()+(R*This.CdimIndex()+C)*sizeof(double);
        
        f.seekg(temp, ios::beg);
        f.read((char*)&data, sizeof(double));
        f.close();
        return data;
    }
    else
    {
        cout<<" WARNING: The matrix does not exist!"<<endl;
        return 0;
    }
VECTOR & MATRIX::operator [] (int k) const
{
    if(k < 0 || k >= Rdim)
        cout << "index " << k << " of matrix is out of range, it should be [0," << Rdim - 1 << "]" << endl;
    return Row[k];
}

MATRX & MATRIX::operator=(const MATRIX& Source)
{
    if(Source.Rdim==0 && Source.CdimIndex()==0)
    {
        cout << "A NULL source matrix" << endl;
    }
    else if(Source.RdimIndex() == Rdim && Source.CdimIndex() == Cdim)
    {
        Copy(Source);
    }
    else
    {
        cout << "Warning: The source matrix has different dimensions." << endl;
        SetUp(Source.RdimIndex(), Source.CdimIndex());
        Copy(Source);
    }
    return This;
}

MATRX & MATRIX::operator+=(const MATRIX& Source)
{
    int i;
    if(Source.Rdim==0 && Source.CdimIndex()==0)
    {
        cout << "Add a NULL source matrix" << endl;
    }
    else if(Source.RdimIndex()!= Rdim || Source.CdimIndex()!= Cdim)
    {
        cout << "Two matrices have different dimensions" << endl;
    }
else
{
    for (i = 0; i < Rdim; i++)
        Row[i] += Source[i];
}
return This;
}

/**
 * MATRIX& MATRIX::operator-=(const MATRIX& Source)
 */
int i;
if(Source.Rdim==0 & Source.CdimIndex()==0)
{
    cout<< "Add a NULL source matrix"<< endl;
}
else if( Source.RdimIndex()!= Rdim || Source.CdimIndex()!= Cdim)
{
    cout<< "Two matrices have different dimensions""<< endl;
}
else
{
    for (i = 0; i < Rdim; i++)
        Row[i] -= Source[i];
}
return This;
}

/**
 * MATRIX& MATRIX::operator*=(const double& x)
 */

for (i = 0; i < Rdim; i++)
    Row[i] *= x;
return This;
}

/**
 * MATRIX operator+(const MATRIX& V, const MATRIX& W)
 */
MATRIX Sum(V);
if((V.RdimIndex()==0 & V.CdimIndex()==0) ||
    (W.RdimIndex()==0 & W.CdimIndex()==0))
    cout<< "One of the matrices is a NULL MATRIX""<< endl;
else if( V.RdimIndex()!=W.RdimIndex() ||
V.CdimIndex() != W.CdimIndex()
    cout << "Two matrices have different dimensions" << endl;
else
    Sum += W;
return Sum;
}

/*****************************/
MATRIX operator-(const MATRIX& V, const MATRIX& W)
/*****************************/
{
    MATRIX Sum(V);
    if((V.RdimIndex()==0 && V.CdimIndex()==0) ||
        (W.RdimIndex()==0 && W.CdimIndex()==0))
        cout << "One of the matrices is a NULL MATRIX" << endl;
    else if( V.RdimIndex() != W.RdimIndex() ||
        V.CdimIndex() != W.CdimIndex())
        cout << "Two matrices have different dimensions" << endl;
    else
        Sum -= W;
return Sum;
}

/*****************************/
MATRIX operator*(const MATRIX& V, const MATRIX& W)
/*****************************/
{
    int i, j, k;
    if(V.CdimIndex() != W.RdimIndex())
        {
            cout << "Two matrices have different inner index." << endl;
            return MATRIX();
        }
    MATRIX Mul(V.RdimIndex(), W.CdimIndex());
    for(i=0; i< V.RdimIndex(); i++)
        for(j=0; j< W.CdimIndex(); j++)
            for(k=0; k< V.CdimIndex(); k++)
                Mul[i][j] += V[i][k]*W[k][j];
    return Mul;
}

/*****************************/
MATRIX operator*(const double k, const MATRIX& W)
/*****************************/
{
    MATRIX Mul(W);

```cpp
int i, j;

for (i = 0; i < W.RdimIndex(); i++)
    for (j = 0; j < W.CdimIndex(); j++)
        Mul[i][j] *= k;
return Mul;

MATRIX operator/(const MATRIX& W, const double k)
{
    return ((1.0/k) * W);
}

VECTOR operator*(const VECTOR& V, const MATRIX& W)
{
    int i, k;

    if (V.DimIndex() != W.RdimIndex())
    {
        cout << "The Vector and matrix have different inner index." << endl;
        return VECTOR();
    }

    VECTOR Mul(W.CdimIndex());
    for (i = 0; i < W.CdimIndex(); i++)
    {
        Mul[i] = 0.;
        for (k = 0; k < V.DimIndex(); k++)
            Mul[i] += V[k] * W[k][i];
    }
    return Mul;
}

VECTOR operator*(const MATRIX& W, const VECTOR& V)
{
    int i, k;

    if (V.DimIndex() != W.CdimIndex())
    {
        cout << "The Vector and matrix have different inner index." << endl;
        return VECTOR();
    }

    VECTOR Mul(W.RdimIndex());
```

for(i=0; i<W.RdimIndex(); i++)
    {
        Mul[i]=0.;
        for(k=0; k<V.DimIndex(); k++)
            Mul[i] += V[k]*W[i][k];
    }
return Mul;
}

MATRIX operator|(const MATRIX& V, const MATRIX& W)
{
    int i,j;
    if(V.RdimIndex()!=W.RdimIndex())
    {
        cout<<"Two matrices have different rows."
    }
    return MATRIX();
}

MATRIX Sum(V.RdimIndex(),V.CdimIndex()+W.CdimIndex());
for(i=0; i<V.RdimIndex(); i++)
    {
        for(j=0; j<V.CdimIndex(); j++)
            Sum[i][j] = V[i][j];
        for(j=0; j<W.CdimIndex(); j++)
            Sum[i][j+W.CdimIndex()] = W[i][j];
    }
return Sum;
}

MATRIX operator&(const MATRIX& V, const MATRIX& W)
{
    int i,j;
    if(V.CdimIndex()!=W.CdimIndex())
    {
        cout<<"Two matrices have different columns."
    }
    return MATRIX();
}

MATRIX Sum(V.RdimIndex()+W.RdimIndex(),V.CdimIndex());
```cpp
for(i=0; i<V.RdimIndex(); i++)
{
    for(j=0; j<V.RdimIndex(); j++)
        Sum[i][j] = V[i][j];

    for(j=0; j<W.RdimIndex(); j++)
        Sum[i+V.RdimIndex()][j] = W[i][j];
}

return Sum;
}

int operator==(const MATRIX & s1, const MATRIX & s2)
{
    if(s1.RdimIndex() == s2.RdimIndex() &&
        s1.CdimIndex() == s2.CdimIndex())
    {
        for (int i = 0; i < s1.RdimIndex(); i++)
            if(s1[i] != s2[i])
                return False;
        return True;
    }
        return False;
}

int operator!=(MATRIX & s1, MATRIX & s2)
{
    if(s1 == s2)
        return False;
    else
        return True;
}

MATRIX & operator*=(MATRIX & V, const MATRIX & W)
{
    return V = V*W;
}

ostream & operator<<(ostream & s, const MATRIX & m)
{
}
```

int i, j;

Form;

if(m.RdimIndex() == 1)
{
    s<<"ln This is a " "1" " vector":
    if (m.get_Name())[0] != NULL)
        s<<", named as " <<m.get_Name()<<"";
    s<<endl;
    s<<" The vector is:"<<endl;
}
else if(m.CdimIndex() == 1)
{
    s<<"ln This is a 1" "1" vector":
    if (m.get_Name())[0] != NULL)
        s<<", named as " <<m.get_Name()<<"";
    s<<endl;
    s<<" The vector is:"<<endl;
}
else
{
    s<<"ln This is a " "1" " matrix":
    if (m.get_Name())[0] != NULL)
        s<<", named as " <<m.get_Name()<<"";
    s<<endl;
    s<<" The matrix is:"<<endl;
    }

Form= m.get_form();
if (Form <=0 || Form >10) // default value for m.form
    Form= 6;
for (i=0; i<m.RdimIndex(); i++)
    {
    s<< endl;
    for (j=0; j<m.CdimIndex(); j++)
        {
        if (Form==1)
            s.setf(ios::scientific);
        else
            s.width(Form);
        s<<m[i][j]<<" ";
    }
    s<< endl;
    return s;
}
MATRIX inverse(const MATRIX& m)

{
    int i,j,k;
    double temp;
    int n= m.CdimIndex();
    MATRIX M(n,n);

    int n2= n*2;

    // Replace original matrix [m1] and Identity matrix [T] with
    //temporary matrix [BB]= [m1][T]
    MATRIX BB(n,n2);
    MATRIX T(n,n);
    T= T.MakeUni();
    BB= m|T;

    // Make values of K column to be 1.0
    for (k= 0; k< n; k++)
    {
        for (i= 0; i< n; i++)
        {
            temp= BB[i][k];
            if (temp != 0.0)
                for (j= 0; j< n2; j++)
                    BB[i][j] /=temp;
        }

        // Subtract all rows by K row
        for (i= 0; i< n; i++)
            if(i !=k&& BB[i][k] !=0.0)
                for(j= k;j<n2; j++)
                    BB[i][j] -=BB[k][j];
    }

    // check if diagnol item of [BB] isn't 1.0. then change to 1.0
    for (i= 0; i< n; i++)
    {
        if(BB[i][i]!=1.0)
        {
            temp= BB[i][i];
            for (j= 0; j< n2; j++)
                BB[i][j] /= temp;
        }
    }

    // Replace temporary matrix with original one
    for (i= 0; i< n; i++)
        for (j= 0; j< n; j++)
            M[i][j]= BB[i][j+n];
return M;
}

/***/
MATRIX trans(const MATRIX& Source)
/***/
{
    int i, j;
    int m = Source.CdimIndex();
    int n = Source.RdimIndex();
    MATRIX temp(m, n);
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            temp[i][j] = Source[j][i];
    return temp;
}

DDI::DDI(int mm, int nn)
{
    int i, j;
    m = mm;
    n = nn;
    array = (int **) malloc(sizeof(int *) * m);
    for (i = 0; i < m; i++)
        array[i] = (int *) malloc(sizeof(int) * n);
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            array[i][j] = 0;
}

#endif

B.3 Source Code of SPARSE_MATRIX.H

#ifndef SPARSE_MATRIX_H
#define SPARSE_MATRIX_H

/***/
Definition of class SPARSE_MATRIX

File name: smatrix.h 1/2/93

Coded by George G. Yu (The Ohio State University)

Compiled by IBM C++ compiler XLC
**include "general.h"
#include "svector.h"
#include "matrix.h"

class SPARSE_MATRIX : public MATRIX
{
 public:
 SPARSE_MATRIX(int, int);
 SPARSE_MATRIX(void);
~SPARSE_MATRIX(void);
 SPARSE_MATRIX& SetUp(int, int);
 SPARSE_MATRIX& SetUp(int,int,int,short*);

 // overloaded operator
 SPARSE_VECTOR& operator ()(int) const;

 friend ostream& operator<<(ostream&, const SPARSE_MATRIX&);

 protected:
 SPARSE_VECTOR *Row;
};

#endif

---

**include "general.h"
#include "svector.h"
#include "matrix.h"

class SPARSE_MATRIX : public MATRIX
{
 public:
 SPARSE_MATRIX(int, int);
 SPARSE_MATRIX(void);
~SPARSE_MATRIX(void);
 SPARSE_MATRIX& SetUp(int, int);
 SPARSE_MATRIX& SetUp(int,int,int,short*);

 // overloaded operator
 SPARSE_VECTOR& operator ()(int) const;

 friend ostream& operator<<(ostream&, const SPARSE_MATRIX&);

 protected:
 SPARSE_VECTOR *Row;
};

#endif

---

### B.4 Source Code of SPARSE_MATRIX.C

```c
#include "smatrix.h"

#ifndef SMATRIX_C
#define SMATRIX_C

/*****************************/
Definition of class SPARSE_MATRIX

File name: smatrix.C 1/2/93

Coded by George G. Yu (The Ohio State University)

Compiled by IBM C++ compiler XLC

/*****************************/
#include "svector.h"
```

```cpp
SPARSE_MATRIX::SPARSE_MATRIX(int N, int M)
   : MATRIX()
{
   SetUp(N, M);
}

SPARSE_MATRIX::SPARSE_MATRIX(void)
   : MATRIX()
{
   cout "enter a null constructor of sparse matrix" << endl;
}

SPARSE_MATRIX::~SPARSE_MATRIX(void)
{
   if(Row != NULL)
      delete [] Row;
}

SPARSE_MATRIX& SPARSE_MATRIX::SetUp(int RowHi, int ColHi)
{
   Rdim = RowHi;
   Cdim = ColHi;
   Row = new SPARSE_VECTOR[Rdim];
   if(RowHi > 0 && Row == NULL)
   {
      SetError(OutOfMemory);
      Rdim = 0;
   }

   Type = OSMATRIX;
   return This;
}

SPARSE_MATRIX& SPARSE_MATRIX::SetUp(int k, int dim, int fill, short*Fill_in)
```
{  
    This[k].setUp(dim, fill, Fill_in);  
    return This;  
}

/*SPARSE_VECTOR & SPARSE_MATRIX::operator[] (int k) const*/
{  
    if(k<0 || k>= Rdim)  
        cout<<"Index " << k << " of sparse matrix is out of range, it should be " <<  
            "[0, " << Rdim-1 << "]" << endl;  
    return Row[k];  
}

/*ostream & operator<<(ostream & s, const SPARSE_MATRIX & m)*/
{  
    int i, j, Form;  
    if(m.RdimIndex()== 1)  
        {  
            s<<"This is a " << m.RdimIndex() << " * 1 sparse vector";  
            if( (m.get_Name())[0] != NULL)  
                s<<", named as " << m.get_Name() << ";" << endl;  
            s<<"The sparse vector is:" << endl;  
        }  
    else  
        {  
            s<<"This is a " << m.RdimIndex() << " * " << m.CdimIndex()  
               << " sparse matrix";  
            if( (m.get_Name())[0] != NULL)  
                s<<", named as " << m.get_Name() << "\n\n";  
            s<<\n\n"The sparse matrix is:" << endl;  
        }  
    Form= m.get_form();  
    if (Form <=0 || Form >10) // default value for m.form  
        Form= 6;  
    for (i=0; i<m.RdimIndex(); i++)  
        {  
            
}
267

s « endl;
for (j=0; j<m.CdimIndex(); j++)
if (Form— 1)
s.setf(ios::scientific);
else
s.width(Form);
s « m [i]Ijl« "
!;
s « endl;
return s;
}
#endif


APPENDIX C

CLASS LIBRARY OF OBJECT-ORIENTED FINITE ELEMENT ANALYSIS

C.1 Source Code of GENERAL.H

```c
#ifndef general_h
#define general_h

/* Definition of general head file */

#include <errno.h>
#include <fcntl.h>
#include <fstream.h>
#include <iostream.h>
#include <math.h>
#include <stdarg.h>
#include <stdlib.h>
#include <stream.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/timeb.h>
#include <sys/types.h>
#include <unistd.h>

extern int errno;
```

Coded by George G. Yu (The Ohio State University)

Compiled by "<AT&T C++ Translator 2.1.03 08/31/90>"
#define SEQ_QKEY (key_t)0104
#define NODE_QKEY (key_t)0105

#define MAT_SHM_KEY (key_t)0x101
#define GD_SHM_KEY (key_t)0x102
#define VOM_SHM_KEY (key_t)0x103
#define SEM_SHM_KEY1 (key_t)0x201
#define SEM_SHM_KEY2 (key_t)0x202
#define STIFF_SHM_KEY (key_t)0x203
#define SEM_KEY1 (key_t)0x301
#define SEM_KEY2 (key_t)0x302

#define QPERM 0660
#define MAXOBN 64
#define MAXPRIOR 50 // it controls the highest No.
    // of node and element can be named
#define SEMPERM 0600

#define This (*this)
#define True 1
#define False 0
#define ZERO 0
#define YES 1
#define OFF 0

typedef enum ErrorType
{
    NoError,
    BadFormat,
    IndexError,
    OutOfMemory,
    DivideError
};

typedef enum Bool
{
    FALSE,    // 0
    TRUE,     // 1
};

typedef enum Object_type
{
    NOTHING,  // 0
    OVECTOR,  // 1
};
OMATRIX,                  // 2
OGENERAL_DATA,             // 3
OSHAPE,                   // 4
OSHAPE_4,                  // 5
OSHAPE_6,                  // 6
OJACOB,                   // 7
ONODE,                    // 8
OMATERIAL,                 // 9
OGAUSS,                   // 10
OELEMENT,                  // 11
OELEMENT_SEQ,              // 12
OBOUCOND,                  // 13
OGLOBAL_DATA,              // 14
OGLOBAL_ELEMENT,           // 15
OLOCAL_ELEMENT,            // 16
OELEMENT_TYPE              // 17
);

int imax(int x, int y);
int imin(int x, int y);
double dmax(double x, double y);
double dmin(double x, double y);
void SetError(ErrorType E);
ErrorType GetError();
char * ErrorName();

#endif
C.2 Source Code of GENERAL.C

#include "general.h"

ErrorType ErrorSignal= NoError;

char * ErrorString[] =
{
    "NoError",
    "BadFormat",
    "IndexError",
    "OutOfMemory",
    "DivideError"
};

char * Object_string[] =
{
    "NOTHING",   // 0
    "VECTOR",    // 1
    "MATRIX",    // 2
    "GENERAL_DATA",   // 3
    "SHAPE",     // 4
    "SHAPE_4",   // 5
    "SHAPE_6",   // 6
    "JACOB",     // 7
    "NODE",      // 8
    "MATERIAL",  // 9
    "GAUSS",     // 10
    "ELEMENT",   // 11
    "ELEMENT_SEQ",  // 12
    "BOUCOND",   // 13
    "GLOBAL_DATA", // 14
    "GLOBAL_ELEMENT", // 15
    "LOCAL_ELEMENT", // 16
    "ELEMENT_TYPE" // 17
};


int imax(int x, int y)
{
    return (x<y? y:x);
};

int imin(int x, int y)
{
    return (x<y? x:y);
};

double dmax(double x, double y)
{
    return (x<y? y:x);
};

double dmin(double x, double y)
{
    return (x<y? x:y);
};

void SetError(ErrorType E)
{
    ErrorSignal= E;
};

ErrorType GetError()
{
    return ErrorSignal;
};

char* ErrorName()
{
    ErrorType E= ErrorSignal;
    ErrorSignal= NoError;
    return ErrorString [E];
};

struct q_dentry
{
    long mtype;
    double mtext[MAXOBN+1];
};

struct q_ientry
{
    long mtype;
    int mtext[MAXOBN+1];
};
C.3 Source Code of FEM.H

#ifndef FEM_H
#define FEM_H

/* CLASS LIBRARY FOR FINITE ELEMENT METHOD */

File name: fem.h 03/10/92

Coded by George G. Yu (The Ohio State University)

Compiled by «AT&T C++ Translator 2.1.03 08/31/90>>

#include "general.h"
#include "vector.h"
#include "matrix.h"

#define inter_node_num 8
#define gauss_point_number 3

// Constants
const A_ele_type = 10;
const A_total_node = 50;

// following classes are included

class GENERAL_DATA;
class SHAPE;
class SHAPE_4;
class SHAPE_6;
class JACOB;
class NODE;
class MATERIAL;
class GAUSS;
class ELEMENT;
class BOUCOND;


class GENERAL_DATA
{public:
   GENERAL_DATA(char*);
};
GENERAL_DATA{};
GENERAL_DATA(const GENERAL_DATA&);
~GENERAL_DATA{};

void input_info(void);
void check_info(void);

int get_Num_nodal_point() const {return Num_nodal_point;};
int get_Num_element() const {return Num_element;};
int get_Num_bc_point() const {return Num_bc_point;};
int get_Num_ele_node() const {return Num_ele_node;};
int get_Num_gauss() const {return Num_gauss;};
int get_Node_DOF() const {return Node_DOF;};
int get_Ele_DOF() const {return Ele_DOF;};
int get_Total_DOF() const {return Total_DOF;};
int get_Num_ele_gauss() const {return Num_ele_gauss;};
int get_Dim_xyz() const {return Dim_xyz;};
int get_Num_ele_type() const {return Num_ele_type;};
int* get_ele_type() const {return Ele_type;};
int* get_Node_freq() const {return Node_freq;};

void assign_Node_DOF(int);
void assign_Ele_type(int *);
void assign_Node_freq(int node_num) {Node_freq[node_num]++};
void init_Node_freq();
void def_DOF(void);

protected:
int Num_nodal_point;
int Num_element;
int Num_bc_point;
int Num_ele_node;
int Num_gauss;
int Node_DOF;
int Ele_DOF;
int Total_DOF;
int Num_ele_gauss;
int Dim_xyz;
int Num_ele_type;
int Ele_type[A_ele_type];
char Title[30];
int Node_freq[A_total_node];
};
public:
SHAPE(int);
~SHAPE(void){};

VECTOR get_shape(void) const {return shape;};
MATRIX get_derive(void) const {return derive;};
int get_node(void) const {return node_num;};
virtual void definition(double ss, double tt)=0;  // virtual base class

protected:
int node_num;
VECTOR shape;
MATRIX derive;

};

class SHAPE_4: public SHAPE
{
public:
SHAPE_4(void): SHAPE(4){};
~SHAPE_4(void){};

void definition(double, double);

protected:
double s, t;

};

class SHAPE_6: public SHAPE
{
public:
SHAPE_6(void): SHAPE(6){};
~SHAPE_6(void){};

void definition(double, double);

protected:
double s, t;

};
class JACOB

public:
JACOB(int, MATRIX&);
~JACOB(void){};

void definition(MATRIX&);
MATRIX get_card(void) const { return cartd;};
double get_djacb(void) const { return djacb;};

protected:
int node_num;
double djacb;
MATRIX elcod,
cartd,
xjaci,
xjacm;

};

class NODE

public:
NODE();
NODE(int, int, int);
NODE(int, int, int, double *);
NODE(int, int, int, double ...);
NODE(const NODE&);
NODE(const NODE&, const NODE&);
NODE(const NODE&, const NODE&,
const int);
~NODE(void){};

void SetUp(int, int, int);
int get_NodeNum(void) const { return NodeNum;};
int get_NumDOF(void) const { return NumDOF;};
int get_NumDim(void) const { return NumDim;};
int get_Status(void) const { return Status;};
int get_Disp_data(void)const {return Disp_data;};
Bool get_ACTIVE(void) const {return ACTIVE;};
char* get_Name(void) const {return Name;};
char* get_File_name(void) const {return File_name;};
VECTOR get_xyz(void) const {return xyz;};
VECTOR get_disp(void) const {return disp;};

streampos get_Offset(void) const {return Offset;};
streampos get_Length(void) const {return Length;};
streampos get_Length_xyz(void) const {return Length_xyz;};
streampos get_Length_disp(void) const {return Length_disp;};

void assign_xyz(double *val);
void assign_disp(double *val);
void assign_xyz(VECTOR&);
void assign_disp(VECTOR&);
void assign_xyz(void);
void assign_status(int i) {Status= i;};
void assign_node_num(int i) {NodeNum= i;};
void assign_Num_Dim(int i) {NumDim= i; };;
void assign_Disp_data(int i) {Disp_data= i;};
friend ostream& operator<<(ostream& , const NODE&);

Bool Store(char*, streampos); // position of matrix head
Bool Retrieve(char*, streampos); // position of matrix head
Bool Delete(char*, streampos); // Entire matrix
streampos Insert(char*, streampos); // Entire matrix
Bool Update_xyz(char*, streampos, VECTOR);
Bool Update_disp(char*, streampos, VECTOR);
Bool Update_xyz(char*, streampos, double*);
Bool Update_disp(char*, streampos, double*);

NODE& Copy(const NODE&);
NODE& operator+=(const NODE&);
NODE& operator =(const NODE&);

protected:
// storage information
 Object_type Type;
 Bool ACTIVE;
streampos Offset; // position of xyz[0]
streampos Length; // =Length_xyz+Length_disp
streampos Length_xyz; // including xyz head
streampos Length_disp;
char File_name[15];

int NodeNum,
 NumDOF,
 NumDim,
 Status, //1: normal node; 2: incremental node;
 Disp_data; // indicate whether disp is assigned
char Name[15];
VECTOR xyz;
VECTOR disp;

};

// Non-member function of Class NODE
NODE operator+(const NODE&, const NODE&);
class ELEMENT_SEQ

public:
ELEMENT_SEQ();
ELEMENT_SEQ(int, int);
ELEMENT_SEQ(int, int, int *);
ELEMENT_SEQ(int, int, int ...);
ELEMENT_SEQ(const ELEMENT_SEQ&);
ELEMENT_SEQ(const ELEMENT_SEQ&, const ELEMENT_SEQ&, const int);
~ELEMENT_SEQ(void){};

void SetUp(int, int);

int get_Ele_Num(void) const {return Ele_Num;};
int get_Num_Node(void) const {return Num_Node;};
int get_Type(void) const {return Ele_type;};
int* get_Seq(void) const {return Seq;};
Bool get_ACTIVE(void) const {return ACTIVE;};
char* get_Name(void) const {return Name;};
char* get_File_name(void) const {return File_name;};
streampos get_Length(void) const {return Length;};
streampos get_Offset(void) const {return Offset;};
Object_type get_Object_type() const {return Type;};

void assign_Seq(void);
void assign_Seq(int*);
void assign_Ele_Num(int i) {Ele_Num = i;};
void assign_Type(int i) {Ele_type = i;};
void assign_status(int i) {Status = i;};

friend ostream& operator<<(ostream&, const ELEMENT_SEQ&);

Bool Store(char*, streampos);
Bool Retrieve(char*, streampos);
Bool Delete(char*, streampos);
Bool Update(int);
streampos Insert(char*, streampos);

ELEMENT_SEQ& Copy (const ELEMENT_SEQ&);
ELEMENT_SEQ& operator+=(const ELEMENT_SEQ&);
ELEMENT_SEQ& operator=(const ELEMENT_SEQ&);

protected:
// storage information
Object_type Type;
Bool ACTIVE;
streampos Offset;  // head position of the record
streampos Length;
char File_name[15];

int Seq[8],
   Ele_Num,
   Num_Node,
   Ele_type,
   Status; //1: normal element; 2: incremental element;
char Name[15];

// Non-member function of Class ELEMENT_SEQ
ELEMENT_SEQ operator+(const ELEMENT_SEQ&, const ELEMENT_SEQ&);

class GAUSS
{
public:
GAUSS(int);
double *get_weight(void) const {return Weight;};
double *get_position(void) const {return Position;};
int get_gpnum(void) const {return gp_num;};
~GAUSS();

protected:
int gp_num;
double Weight[gauss_point_number*gauss_point_number],
       Position[gauss_point_number*gauss_point_number];

private:
void def_gauss(void);
};

class ELEMENT : public GENERAL_DATA, public GAUSS
{
public:
ELEMENT(int, int, GENERAL_DATA&);
ELEMENT(int, int, int, int, int, int, int, int);
~ELEMENT();

void def_Inods(streampos*);
void def_Elcod(streampos*);
virtual void definition(void)=0;

int* get_Lnods(void) const {return Lnods;};
int get_Ele_Num(void) const {return Ele_Num;};
MATRIX get_Stiff(void) const {return Stiff;};

protected:
int Lnods[inter_node_num],
Ele_Type,
Ele_Num,
Num_Node,
Num_Total_Point.
Dim_xyz,
Num_Gauss,
Num_Element,
Num_Nodal_DOF,
Num_DOF;
double Dvolu;
VECTOR Shape;
MATRIX Stiff,
    Gpcod,
    Derive,
    Elcod,
    Cartd;

};

class BCOND
class BCOND

{ public:
    BCOND(int,int,int);
~BCOND(void){{};
void assign_bc(void);
DDI& get_Bc(void) {return Be;};
MATRIX& get_BcVal(void) {return BcVal;};

protected:
int TotalNodeNum,
    NumBc,
    NumDim;
DDI Bc;
MATRIX BcVal;

// Bc[][]= 0 -> unknown; Bc[][]= 1 -> known disp.; Bc[][]= 2 -> known force:
};
#endif //__FEM_H__
C.4 Source Code of FEM.C

#include "fem.h"

SHAPE::SHAPE(int m) : shape(m), derive(2,m) {
    node_num = m;
}

void SHAPE_4::definition(double ss, double tt) {
    s = ss;
    t = tt;
    double st = s*t;
    shape[0] = (1-t-s+st)*.25;
    shape[1] = (1-t+s-st)*.25;
    shape[2] = (1+t+s+st)*.25;
    shape[3] = (1+t-s-st)*.25;
    derive[0][0] = (-1-t)*.25;
    derive[0][1] = (1-t)*.25;
    derive[0][2] = (1+t)*.25;
    derive[0][3] = (-1-t)*.25;
    derive[1][0] = (-1+s)*.25;
    derive[1][1] = (-1-s)*.25;
    derive[1][2] = (1+s)*.25;
    derive[1][3] = (1-s)*.25;
}

void SHAPE_6::definition(double ss, double tt) {
    s = ss;
    t = tt;
    double st = s*t;
    shape[0] = (1-t-s+st)*.25;
    shape[1] = (1-t+s-st)*.25;
    shape[2] = (1+t+s+st)*.25;
    shape[3] = (1+t-s-st)*.25;
    shape[4] = (1+t+s-st)*.25;
    shape[5] = (1+t-s-st)*.25;
    derive[0][0] = (-1-t)*.25;
    derive[0][1] = (1-t)*.25;
    derive[0][2] = (1+t)*.25;
    derive[0][3] = (-1-t)*.25;
    derive[0][4] = (1+s)*.25;
    derive[0][5] = (-1+s)*.25;
    derive[1][0] = (-1+s)*.25;
    derive[1][1] = (-1-s)*.25;
    derive[1][2] = (1+s)*.25;
    derive[1][3] = (1-s)*.25;
    derive[1][4] = (-1+s)*.25;
    derive[1][5] = (-1-s)*.25;
}
\{ 
  s = ss;
  t = tt;
  double st = s*t, u = 1 - s - t;

  shape[0] = -u*(1 - 2.0*u);
  shape[1] = 4.0*s*u;
  shape[2] = -s*(1.0 - 2.0*s);
  shape[3] = 4.0*st;
  shape[4] = -t*(1.0 - 2.0*t);
  shape[5] = 4.0*t*u;

  derive[0][0] = 1.0 - 4.0*u;
  derive[0][1] = 4.0*(u - s);
  derive[0][2] = -1.0 + 4.0*s;
  derive[0][3] = 4.0*t;
  derive[0][4] = 0.0;
  derive[0][5] = -4.0*t;
  derive[1][0] = 1.0 - 4.0*u;
  derive[1][1] = -4.0*s;
  derive[1][2] = 0.0;
  derive[1][3] = 4.0*s;
  derive[1][4] = -1.0 + 4.0*t;
  derive[1][5] = 4.0*(u - t);
\}

/******************************************************************************
JACOB: JACOB(int m, MATRIX& Elcod)
:elcod(Elcod), cartd(2,m), xjaci(2,2), xjacm(2,2)
/******************************************************************************/
{ 
  node_num = m;
}

void JACOB::definition(MATRIX& derive) 
/******************************************************************************/
{ 
  // cout<<"in Jacob::elcod:"<<elcod;

  // Generating Jacobian matrix
  // since elcod[2][inode] and derive[2][inode] having the same dimension,
  // transversing elcod is necessary.
  MATRIX temp = trans(elcod);
  xjacm = derive*temp;

  // Determinant of Jacobian matrix
  djacb = xjacm[0][0]*xjacm[1][1] - xjacm[0][1]*xjacm[1][0];

  // Checking djacb
if(djacb< 0.0)
    cout<< "DJACB< 0.0, Check!!!"<< endl;

    xjaci[0][0] = xjacm[1][1]/djacb;
    xjaci[1][1] = xjacm[0][0]/djacb;
    xjaci[0][1] = xjacm[0][1]/djacb;
    xjaci[1][0] = xjacm[1][0]/djacb;

// Cartesian derivatives
    cartd= xjacl*derive;
}

NODE::NODE(int Num, int Nval, int NDOF)
{/*****/
    SetUp(Num,Nval,NDOF);
    assign_status(1);
}

NODE::NODE(int Num, int Nval, int NDOF, double *val)
{/*****/
    SetUp(Num,Nval,NDOF);
    assign_xyz(val);
    assign_status(1);
}

NODE::NODE(int Num, int Nval, int NDOF, double first ...)
{/*****/
    SetUp(Num,Nval,NDOF);
    double *temp= new double[NumDim];
    temp[0] = first;
    va_start(ap, first);
    for (i=1; i< NumDim; i++)
        temp[i] = double(va_arg(ap, double));
    assign_xyz(temp);
    delete [] temp;
assign_status(1);
}

NODE::NODE(const NODE& N)
/*--------------------*/
{
   SetUp(N.get_NodeNum(), N.get_NumDim(), N.get_NumDOF());
   Copy(N);
}

NODE::NODE(const NODE& N1, const NODE& N2)
/*----------------*/
{
   if(N1.get_NumDim()==N2.get_NumDim() &&
      N1.get_NodeNum()!=N2.get_NodeNum())
   {
      SetUp(N1.get_NodeNum()+N2.get_NodeNum(),N1.get_NumDim(),N1.get_NumDOF());
      assign_status(1);
      This= N1+N2;
   }
   else
   {
      cout<<"Warning: No new node is createdd."<endl;
   }
}

NODE::NODE(const NODE& N1, const NODE& N2, const int Incre)
/*----------------*/
{
   int i;

   if(N1.get_NumDim()==N2.get_NumDim() &&
      N1.get_NodeNum()!=N2.get_NodeNum() &&
      ((N2.get_NodeNum()-N1.get_NodeNum())>=Incre))
   {
      SetUp(0,N1.get_NumDim(),N1.get_NumDOF());
      for(i=0; i<N1.get_NumDim(); i++)
         xyz[i]= ((N2.get_xyz())[i]-(N1.get_xyz())[i])/Incre;
      NodeNum= (N2.get_NodeNum()-N1.get_NodeNum())/Incre;
      assign_status(2);
   }
   else
      cout<<"WARNING: No incremental nodes are created"<endl;
void NODE::SetUp(int Num, int Dim, int NDOF)
/*****************************/
{
    NodeNum= Num;
    NumDim= Dim;
    NumDOF= NDOF;
    Disp_data= NO;
    Type= ONODE;
    xyz.SetUp(Dim);
    disp.SetUp(NDOF);
}

void NODE::assign_xyz(double *val)
/*****************************/
{
    int i;
    for (i=0; i<NumDim; i++)
        xyz[i] = val[i];
}

void NODE::assign_xyz(VECTOR& val)
/*****************************/
{
    int i;
    for (i=0; i<NumDim; i++)
        xyz[i] = val[i];
}

void NODE::assign_disp(double *val)
/*****************************/
{
    int i;
    for (i=0; i<NumDOF; i++)
        disp[i] = val[i];
    Disp_data = YES;
}

void NODE::assign_disp(VECTOR& val)
/*****************************/
{
    int i;
    for (i=0; i<NumDOF; i++)
        disp[i] = val[i];
Disp_data = YES;
}

void NODE::assign_xyz(void)
{
    int i;
    cout << "Please input coordinates of Node #" << NodeNum << " : (" << NumDim << " dimension)" << endl;
    for (i = 0; i < NumDim; i++)
        cin >> xyz[i];
}

ostream& operator<<(ostream& s, const NODE &m)
{
    int i;

    if (m.get_Status() == 1)
        s << "The node #" << m.get_NodeNum() << " is located at (";
    else if (m.get_Status() == 2)
        s << "The increment of node coordinates is: (";

    for (i = 0; i < m.get_NumDim() - 1; i++)
        s << (m.get_xyz())[i] << ", ";
    s << (m.get_xyz())[m.get_NumDim() - 1] << ")" << endl;

    if (m.get_Disp_data() == YES)
    {
        s << "The displacements are (";
        for (i = 0; i < m.get_NumDOF() - 1; i++)
            s << (m.get_disp())[i] << ", ";
        s << (m.get_disp())[m.get_NumDOF() - 1] << ")" << endl;
    }

    s << endl;
    return s;
}

Bool NODE::Store(char* f_name, streampos Start)
{
int size_d = sizeof(double);
int size_n = sizeof(NODE);
fstream f;
streampos Offset_disp;

This.ACTIVE = TRUE;
This.Disp_data = NO;
strcpy(File_name, f_name);

f.open(File_name, ios::in | ios::out);
if(!f)
{
    cout << "Cannot open file " << f_name << endl;
    return FALSE;
}
else
{
    f.seekp(Start, ios::beg);
    f.write((char*) & (This), size_n);
    f.close();
}

This.Offset = This.xyz.Insert(File_name, Start + streampos(size_n));
if(This.Offset != -1)
{
    Length_xyz = xyz.get_Length() + sizeof(VECTOR);
}
else
    cout << "xyz storing failure. " << endl;

Offset_disp = This.disp.Insert(File_name, This.Offset + Length_xyz);
if(Offset_disp != -1)
{
    Length_disp = disp.get_Length() + sizeof(VECTOR);
}
else
    cout << "disp storing failure. " << endl;

Length = Length_xyz + Length_disp;

f.open(File_name, ios::in | ios::out);
if(!f)
{
    cout << "Cannot open file " << f_name << endl;
    return FALSE;
}
else
{
    f.seekp(Start, ios::beg);
    f.write((char*) & (This), size_n);
    f.close();
}
Bool NODE::Retrieve(char* f_name, streampos Start)
/*****************************************************************************/
{
    int size_d = sizeof(double);
    int size_n = sizeof(NODE);
    fstream f;

    f.open(f_name, ios::in);
    if(!f)
    {
        cout<<"Cannot open file "<<f_name<<endl;
        return FALSE;
    }
    else
    {
        f.seekg(Start, ios::beg);
        f.read((char*)&(This), size_n);
        f.close();
    }

    if(This.ACTIVE== TRUE)
    {
        streampos Offset_xyz = This.get_Offset();
        streampos Offset_disp = Offset_xyz + This.get_Length_xyz();

        if(xyz.Retrieve(f_name, Offset_xyz)==FALSE)
            cout<<"xyz retrieving failure"<<endl;

        if(disp.Retrieve(f_name, Offset_disp)==FALSE)
            cout<<"disp retrieving failure"<<endl;

        return TRUE;
    }
    else
    return FALSE;
}

Bool NODE::Delete(char* f_name, streampos Start)
/*****************************************************************************/
{
    int size_n = sizeof(NODE);
    fstream f;

    f.open(f_name, ios::in|ios::out);
if(!f)
{
  cout << "Cannot open file " << f_name << endl;
  return FALSE;
}
else
{
  f.seekg(Start, ios::beg);
  f.read((char*)&(This), size_n);

  if(This.ACTIVE== TRUE)
  {
    This.ACTIVE= FALSE;
    f.seekp(Start, ios::beg);
    f.write((char*)&(This), size_n);
    f.close();
    return TRUE;
  }
  else
  {
    f.close();
    cout << "WARNING: The record is already deleted." << endl;
    return FALSE;
  }
}
}

streampos NODE::Insert(char* f_name, streampos Start)
{************
{
  streampos A_Length, R_Length;
  int size_n= sizeof(NODE);
  fstream f;
  NODE temp;
  streampos New_Start, End;
  int input;

  f.open(f_name, ios::in|ios::out);
  if(!f)
  {
    cout << "Cannot open file " << f_name << endl;
    return -1;
  }
  else
  {
    f.seekg(0, ios::end);
    End= f.tellg();

    if(Start!= -1)
    {

if(End > Start)  // avoiding out of file
{
    f.seekg(Start, ios::beg);
    f.read((char^)&temp, size_n);

    A_Length= temp.get_Length();
    R_Length= (This.xyz.DimIndex()+This.disp.DimIndex())
    *sizeof(double)
    +2*sizeof(VECTOR);
    if(A_Length < R_Length)
        New_Start= End;
    else
    {
        if(temp.get_ACTIVE()== YES)
        {
            cout<<"Are you sure to override the existing record?"
                <<"(0/1) (NO/YES)"<<endl;
            cin >> input;
            if(input == NO)
                New_Start= End;
            else
                New_Start= Start;
        }
        else
            New_Start= Start;
    }
    else
        New_Start= Start;
}
else
    New_Start= End;
}
else
    New_Start= End;
f.close();

if(This.Store(f_name, New_Start)== TRUE)
    return New_Start;
else
    return -1;
}

Bool NODE::Update_xyz(char* f_name, streampos Start, double *d)
/*****************************/
{
    streampos Offset;
    int size_n= sizeof(NODE);
    fstream f;
    NODE temp;

    f.open(f_name,ios::in|ios::out);
if(!f)
{
    cout<<"Cannot open file ":<<f_name<<endl;
    return FALSE;
}
else
{
    f.seekg(Start, ios::beg);
    f.read((char*)&temp, size_n);
    Offset= temp.get_Offset();
    f.close();
    temp.assign_xyz(d);
    Offset= temp.xyz.Insert(f_name, Offset);
    if(Offset== -1)
        cout<<"Update xyz failure"<<endl;
    else
        return TRUE;
}

Bool NODE::Update_xyz(char* f_name, streampos Start, VECTOR V)
✓---------------------------------------------------------------------/
{
    double *d= new double[V.DimIndex()];
    for(int i=0; i<V.DimIndex(); i++)
        d[i]= V[i];
    if(Update_xyz(f_name, Start, d) !=TRUE)
        cout<<"Update xyz failure"<<endl;
    else
        return FALSE;
    delete [] d;
    return TRUE;
}

Bool NODE::Update_disp(char* f_name, streampos Start, double *d)
✓---------------------------------------------------------------------/
{
    streampos Offset;
    int size_n= sizeof(NODE);
    fstream f;
    NODE temp;

f.open(f_name,ios::in|ios::out);
if(!f)
{
    cout<<"Cannot open file "<<f_name<<endl;
    return FALSE;
}
else
{
    f.seekg(Start, ios::beg);
    f.read((char*)temp, size_n);

    Offset= temp.get_Offset()+temp.get_Length_xyz();
    f.close();

    temp.assign_xyz(d);
    Offset= temp.disp.Insert(f_name, Offset);
    if(Offset== -1)
        cout<<"Update disp failure"<<endl;
    else
        return TRUE;
}

Bool NODE::Update_disp(char* f_name, streampos Start, VECTOR V)
/*********************************************************************/
{
    double *d= new double[V.DimIndex()];

    for(int i=0; i<V.DimIndex(); i++)
        d[i]= V[i];

    if(Update_disp(f_name, Start, d )!=TRUE)
        cout<<"Update disp failure"<<endl;
    else
        return FALSE;

    delete [] d;
    return TRUE;
}

NODE& NODE::Copy(const NODE& N)
/*********************************************************************/
{
    int i;

    ...
for(i=0; i<NumDim; i++)
    xyz[i] = (N.get_xyz())[i];

if(N.get_Disp_data() == YES)
    for(i=0; i<NumDOF; i++)
        disp[i] = (N.get_disp())[i];

strncpy(Name, N.get_Name());
strncpy(File_name, N.get_File_name());
Status = N.get_Status();
Disp_data = N.get_Disp_data();
Offset = N.get_Offset();
Length = N.get_Length();
Length_xyz = N.get_Length_xyz();
Length_disp = N.get_Length_disp();
ACTIVE = N.get_ACTIVE();

return This;

NODE& NODE::operator+=(const NODE& DN)
/*****************************/
{
    int i;

    if(NumDim == DN.get_NumDim())
    {
        for(i=0; i<NumDim; i++)
            xyz[i] += (DN.get_xyz())[i];
        NodeNum += DN.get_NodeNum();
    }
    else
    {
        cout<<"Node "+NodeNum<<" is different with Node "+DN.get_NodeNum()<<"."<<endl;
    }

    return This;
}

NODE& NODE::operator=(const NODE& N)
/*********************/
{
    SetUp(N.get_NodeNum(), N.get_NumDim(), N.get_NumDOF());
    Copy(N);
    return This;
}
NODE operator+(const NODE& N1, const NODE& N2)
/***************************************************************************/
{
    int i;
    NODE SUM((N1.get_NodeNum()+N2.get_NodeNum()),N1.get_NumDim(),0);
    double *xyz;
    xyz= new double[N1.get_NumDim()];

    if(N1.get_NumDim()==N2.get_NumDim())
    {
        for(i=0; i<N1.get_NumDim(); i++)
            xyz[i]=(N2.get_xyz())[i]+(N1.get_xyz())[i];
        SUM.assign_xyz(xyz);
    }
    else
    {
        cout<<"Node #"<<N1.get_NodeNum()<<" is different with Node #"<<N1.get_NodeNum()<<". No incremental node is created."
    }
    delete [N1.get_NumDim()]*xyz;
    SUM.assign_status(1);
    SUM.assign_Disp_data(NO);
    return SUM;
}

/***************************************************************************/
ELEMENT_SEQ ELEMENT_SEQ::ELEMENT_SEQ(int Num, int Nnode)
/***************************************************************************/
{
    SetUp(Num, Nnode);
    assign_status(1);
}

ELEMENT_SEQ::ELEMENT_SEQ(int Num, int Nnode, int *seq)
/***************************************************************************/
{
    SetUp(Num, Nnode);
    assign_status(1);
    assign_Seq(seq);
}

ELEMENT_SEQ::ELEMENT_SEQ(int Num, int Nnode, int first...)
/***************************************************************************/
{
    int i;
va_list ap;

SetUp(Num, Nnode);
int *temp= new int[Num_Node];
temp[0] = first;
va_start(ap, first);
for (i=1; i < Num_Node; i++)
    temp[i] = int(va_arg(ap, int));
assign_Seq(temp);
delete [Num_Node] temp;
assign_status(1);
}

ELEMENT_SEQ::ELEMENT_SEQ(const ELEMENT_SEQ& N)
/*******************************************/
{
    SetUp(N.get_Ele_Num(), N.get_Num_Node());
    Copy(N);
}

ELEMENT_SEQ::ELEMENT_SEQ(const ELEMENT_SEQ& N1, const ELEMENT_SEQ& N2,
const int incr)
/*******************************************/
{
    int i;
    if(N1.get_Num_Node() == N2.get_Num_Node())
    {
        SetUp(0, N1.get_Num_Node());
        for (i=0; i<N1.get_Num_Node(); i++)
            Seq[i] = ((N2.get_Seq())[i] - (N1.get_Seq())[i])/incr;
        Ele_Num = (N2.get_Ele_Num() - N1.get_Ele_Num())/incr;
        assign_status(2);
    }
    else
    {
        cout<<"Element ",<<N1.get_Ele_Num()<<" is different with Element ",
        <<N2.get_Ele_Num()<<". No incremental element is created."
<<endl;
    }
}

void ELEMENT_SEQ::SetUp(int Num, int Nnode)
/*******************************************/
{
    Ele_Num = Num;
    Num_Node = Nnode;
    Ele_type = 0;
    Type = OELEMENT_SEQ;
void ELEMENT_SEQ::assign_Seq(int* seq)
/***********************
{
    int i;
    for (i=0; i<Num_Node; i++)
        Seq[i]= seq[i];
}

ostream& operator<< (ostream& s, const ELEMENT_SEQ &m)
/*******************************
{
    int i;
    if(m.get_Status()==1)
    {
        s<<"ln The Element #"<< m.get_Ele_Num()<<" has node of (";
        for(i=0; i<m.get_Num_Node(); i++)
            s<<(m.get_Seq())[i]<<" ";
        if(m.get_Type()!= 0)
            s<<")Its type is #"<<m.get_Type()<<endl;
        else
            s<<")"<<endl;
    }
    else if(m.get_Status()==2)
    {
        s<<"ln The increment of element is: (";
        for(i=0; i<m.get_Num_Node(); i++)
            s<<(m.get_Seq())[i]<<" ";
        s<<")"<<endl;
        s<<"The element number increment is "<<m.get_Ele_Num()<<endl;
    }
    return s;
}

Bool ELEMENT_SEQ::Store(char* f_name, streampos Start)
int size_seq = sizeof(ELEMENT_SEQ);
fstream f;

This.ACTIVE = TRUE;
strncpy(File_name, f_name);

f.open(File_name, ios::in | ios::out);
if(!f)
{
    cout << "Cannot open file " << f_name << endl;
    return FALSE;
}
else
{
    Offset = Start;
    Length = size_seq;

    f.seekp(Start, ios::beg);
    f.write((char*)&(This), size_seq);
    f.close();

    return TRUE;
}

Bool ELEMENT_SEQ::Retrieve(char* f_name, streampos Start)
().'/***********************************************************************'/
{
    int size_seq = sizeof(ELEMENT_SEQ);
    fstream f;

    f.open(f_name, ios::in);
    if(!f)
    {
        cout << "Cannot open file " << f_name << endl;
        return FALSE;
    }
    else
    {
        f.seekg(Start, ios::beg);
        f.read((char*)&(This), size_seq);
        f.close();
    }

    if(This.ACTIVE == TRUE)
    return TRUE;
else
    return FALSE;
}

Bool ELEMENT_SEQ::Delete(char* f_name, streampos Start)
/**
 * ELEMENT_SEQ::Delete
 *
 */
{
    int size_seq= sizeof(ELEMENT_SEQ);
    fstream f;

    f.open(f_name,ios::in|ios::out);
    if(!f)
    {
        cout<<"Cannot open file "<<f_name<<endl;
        return FALSE;
    }
    else
    {
        f.seekg(Start, ios::beg);
        f.read((char*)&(This), size_seq);

        if(This.ACTIVE== TRUE)
        {
            This.ACTIVE= FALSE;
            f.seekp(Start, ios::beg);
            f.write((char*)&(This), size_seq);
            f.close();
            return TRUE;
        }
        else
        {
            f.close();
            cout<<"WARNING: The record is already deleted."<<endl;
            return FALSE;
        }
    }
}

Bool ELEMENT_SEQ::Update(int k)
/**
 * ELEMENT_SEQ::Update
 *
 */
{
    fstream f;

    f.open(File_name,ios::in|ios::out);
    if(!f)
    {
cout<<"Warning: in ELEMENT_SEQ::Updat: Cannot open file "
<<File_name<<endl;
return FALSE;
}
else
{*}
/*
72 is computed from:
streampos pos= Offset+sizeof(Type)+sizeof(ACTIVE)
+sizeof(Offset)+sizeof(Length)+sizeof(File_name)
+sizeof(File_name[15])+sizeof(Seq)+sizeof(Ele_Num)
+sizeof(Num_Node);
=Offset+4+4+4+15+1+4*8+4+4
=Offset+72
*/
streampos pos= Offset+72;
f.seekp(pos, ios::beg);
f.write((char*)&k, sizeof(int));

f.close();
return TRUE;
*
streampos ELEMENT_SEQ::Insert(char* f_name, streampos Start)
/* everlasting *//*
*
streampos A_Length, R_Length:
int size_seq= sizeof(ELEMENT_SEQ);
fstream f;
ELEMENT_SEQ temp;
streampos New_Start, End;
int input;

f.open(f_name,ios::in|ios::out);
if(!f)
{
    cout<<"Cannot open file "<<f_name<<endl;
    return -1;
}
else
{
    f.seekg(0,ios::end);
    End= f.tellg();

    if(Start!= -1)
    {
        if(End> Start) // avoiding out of file
        {
f.seekg(Start, ios::beg);
f.read((char*)&temp, size_seq);

A_Length= temp.get_Length();
R_Length= size_seq;
if(A_Length < R_Length)
    New_Start= End;
else
    {
        if(temp.get_ACTIVE()== YES)
            {
                cout<<"Are you sure to override the existing"
                  <<" record?"
                  <<"(0/1) (NO/YES)"<<endl;
                cin >> input;
                if(input == NO)
                    New_Start= End;
                else
                    New_Start= Start;
            }
        else
            New_Start= Start;
    }
else
    New_Start= Start;
}
else
    New_Start= End;
f.close();

if(This.Store(f_name, New_Start)== TRUE)
    return New_Start;
else
    return -1;
}

ELEMENT_SEQ& ELEMENT_SEQ::Copy(const ELEMENT_SEQ& N)
{***********************
    int i;
    for(i=0; i<Num_Node; i++)
        Seq[i]= (N.get_Seq())[i];
    strcpy(Name, N.get_Name());
    strcpy(File_name, N.get_File_name());

    // Type= N.get_Object_type(); it is assigned in SetUp
 ACTIVE = N.get_ACTIVE();
 Offset = N.get_Offset();
 Length = N.get_Length();

 // Ele_Num = N.get_Ele_Num(); it is assigned in SetUp
 // Num_Node = N.get_Num_Node(); it is assigned in SetUp
 Status = N.get_Status();

 return This;

ELEMENT_SEQ& ELEMENT_SEQ::operator+=(const ELEMENT_SEQ& DN)

{ /* ELEMENT_SEQ& ELEMENT_SEQ::operator+=(const ELEMENT_SEQ& DN) */
  int i;

  if(Num_Node == DN.get_Num_Node())
  { for(i = 0; i < Num_Node; i++)
      Seq[i] += (DN.get_Seq())[i];
      Ele_Num += DN.get_Ele_Num();
  }
  else
  { cout << "Element ", i; <<Ele_Num" <<" is different with Element ", i; <<DN.get_Ele_Num()<<" in node number." <<endl;
  }
  return This;
}

ELEMENT_SEQ& ELEMENT_SEQ::operator=(const ELEMENT_SEQ& N)

{ /* ELEMENT_SEQ& ELEMENT_SEQ::operator=(const ELEMENT_SEQ& N) */
  SetUp(N.get_Ele_Num(), N.get_Num_Node());
  Copy(N);
  return This;
}

ELEMENT_SEQ operator+(const ELEMENT_SEQ& N1, const ELEMENT_SEQ& N2)

{ /* ELEMENT_SEQ operator+(const ELEMENT_SEQ& N1, const ELEMENT_SEQ& N2) */
  int i;
  ELEMENT_SEQ SUM((N1.get_Ele_Num()+N2.get_Ele_Num()),
                  N1.get_Num_Node());

  int *temp;
  temp = new int[N1.get_Num_Node()];

  if(N1.get_Num_Node() == N2.get_Num_Node())
  { for(i = 0; i < N1.get_Num_Node(); i++)
  }
temp[i] = (N2.get_Seq())[i] + (N1.get_Seq())[i];
SUM.assign_Seq(temp);
}
else
{
    cout << "Element #" << N1.get_Ele_Num()
    << " is different with Element #"
    << N1.get_Ele_Num() << " in node number. ";
    cout << "No incremental node is created." << endl;
}
delete [] temp;
SUM.assign_status(1);
return SUM;

GAUSS::GAUSS(int n)
{  
gp_num= n*n;
def_gauss();
}

void GAUSS::def_gauss(void)
{  
    int i, j;
    if (gp_num==2)
    {
        Weight[0]= (double)1.0;
        Position[0]= (double)(-0.577350269189626);
    }
    else
    {
        Weight[0]= (double)0.5555555555555556;
        Weight[1]= (double)0.8888888888888889;
        Position[0]= (double)(-0.774596669241483);
        Position[1]= (double)0.;
    }
    for (i= 0; i< (gp_num/2); i++)
    {
        j= gp_num-i-1;
        Weight[j]= Weight[i];
        Position[j]= -Position[i];
ELEMENT::ELEMENT(int Num, int EDOF, GENERAL_DATA& GD_in) :
GENERAL_DATA(GD_in), GAUSS(GD_in.get_Num_gauss()),
Stiff(EDOF, EDOF),
Gpcod(get_Dim_xyz(), get_Num_ele_node()),
Shape(get_Num_ele_node()), Derive(2, get_Num_ele_node()),
Elcod(get_Dim_xyz(), get_Num_ele_node()),
Cartd(get_Dim_xyz(), get_Num_ele_node())
{
Ele_Num = Num;
Num_Node = GD_in.get_Num_ele_node();
Num_Total_Point = GD_in.get_Num_nodal_point();
Dim_xyz = get_Dim_xyz();
Num_Gauss = get_Num_gauss();
Num_DOF = EDOF;
Num_Element = get_Num_element();
}

void ELEMENT::def_nods(streampos* Pos)
{ int i; int ele_size;
ele_size = sizeof(ELEMENT_SEQ):

ELEMENT_SEQ El;
/*
streampos fcount;
fstream fin;
fin.open("ele.dat",ios::in);
if(!fin)
{
    cout<<"Cannot open for output"<<endl;
    exit(-1);
}
fcount = 0;
*/
for (i = 0; i < Num_Element; i++)
{
    /*
    fin.seekg(fcount,ios::beg);
    fin.read((char*)&El, ele_size);
    fcount = fin.tellg();
    */
    El.Retrieve("try.dat",Pos[i]);
    if(Ele_Num== El.get_Ele_Num())
    {
        for(j = 0; j < Num_Node; j++)
            Lnods[j] = (El.get_Seq())[j];
        Ele_Type = El.get_Type();
        i = Num_Element;
    }
}
// fin.close();

void ELEMENT::def_Elcod(streampos *Pos)
{*******************************/
    int i,j, m;
    NODE node;
    /*
    int node_size;
    streampos Node_length;
    node_size = sizeof(NODE);
    Node_length = node_size + (Dim_xyz+Num_Nodal_DOF)*sizeof(double) +sizeof(VECTOR)*2;
    */
for (i=0; i<Num_Total_Point; i++)
{
    node.Retrieve("try.dat",Pos[i]);

    for(j=0; j<Num_Node; j++)
    if(node.get_NodeNum()==Lnods[j])
    {
        for(m=0; m<Dim_xyz; m++)
            Elcod[m][j]= (node.get_xyz())[m];
    }
}

BCOND::BCOND(int totalnodenum, int numbc, int numDim)
:Bc(numbc,numDim), BcVal(numbc,numDim)
{/****

TotalNodeNum= totalnodenum;
NumBc= numbc;
NumDim= numDim;

*****/
void BCOND::assign_bc(void)
{/****

*****/
{  
    int i,j,
        NumBcType,
        NType,
        Nrestr,
        nodenum,
        bctype;
    double **BcTypeVal;
    int ** BcType;

    cout<< "How many boundary condition types?"<< endl;
    cin>> NumBcType;
    BcType= (int **) malloc(sizeof(int*) *NumBcType);
    BcTypeVal= (double **) malloc(sizeof(double *) *NumBcType);

    for (i=0; i<NumBcType; i++)
    {
        cout<< "Input B.C. type number:";
        cin>>NType;
        cout<<"How many restrains:";
        cin>>Nrestr;
        BcTypeVal[NType-1]= (double *) malloc(sizeof(double *)
BcType[NT-1] = (int *) malloc(sizeof(int *) * (Nrestr+1));
BcType[NT-1][0] = Nrestr;

cout<<"Restriction type and B.C. values:"<<endl;
for (j=0; j<Nrestr; j++)
{
    cout<<"Restrain #"<<j+1<<" : ";
    cin>> BcType[NT-1][j+1];
    if (BcType[NT-1][j+1]==0)
        cout<<"the variable is unknown"<<endl;
    else
        cin>> BcTypeVal[NT-1][j];
}

cout<<"Please Input Node Number and its B.C. Type "<<endl;
for (i=0; i< TotalNodeNum; i++)
{
    cin>> nodenum>>bctype;
    Bc.array[nodenum-1][0] = BcType[bctype-1][0];
    for (j=0; j<BcType[bctype-1][0]; j++)
    {
        Bc.array[nodenum-1][j+1] = BcType[bctype-1][j+1];
        BcVal[nodenum-1][j] = BcTypeVal[bctype-1][j];
    }
}

GENERAL_DATA::GENERAL_DATA(char *name)
/*****************************/
{
    init_Node_freq();
    strcpy(Title, name);
}

GENERAL_DATA::GENERAL_DATA(const GENERAL_DATA& S)
/*****************************/
{
    Num_nodal_point= S.Num_nodal_point;
    Num_element= S.Num_element;
    Num_bc_point= S.Num_bc_point;
    Num_ele_node= S.Num_ele_node;
    Num_gauss= S.Num_gauss;
    Node_DOF= S.Node_DOF;
    Total_DOF= S.Total_DOF;
    Num_ele_gauss= S.Num_ele_gauss;
    Dim_xyz= S.Dim_xyz;


```cpp
void GENERAL_DATA::input_info(void)
{
    cout << "\tt \"Title\":\n" << endl;
    cout << "\tt \=";
    for (int i = 0; i < strlen(Title) + 1; i++) cout << "=";
    cout << endl;
    cout << "\tt Please input the following general information.\n" << endl;
    cout << "\tt Total Nodal Point: \n";
    cin >> Num_nodal_point;
    cout << "\tt Total Number of Element: \n";
    cin >> Num_element;
    cout << "\tt Total Number of Node Having B.C.: \n";
    cin >> Num_bc_point;
    cout << "\tt Number of Node at Each Element: \n";
    cin >> Num_ele_node;
    cout << "\tt Number of Gauss Point: \n";
    cin >> Num_gauss;
    cout << "\tt Dimension of Coordinates: \n";
    cin >> Dim_xyz;

    Num_ele_gauss = Num_gauss * Num_gauss;
    Ele_DOF = Node_DOF * Num_ele_node;
}

void GENERAL_DATA::check_info(void)
{
    cout << "\n\tReview the General Information Provided.\n" << endl;
    cout << "\tt Total Nodal Point: \n" << Num_nodal_point << endl;
    cout << "\tt Total Number of Element: \n" << Num_element << endl;
    cout << "\tt Total Number of Node Having B.C.: \n" << Num_bc_point << endl;
    cout << "\tt Number of Node at Each Element: \n" << Num_ele_node << endl;
    cout << "\tt Number of Gauss Point: \n" << Num_gauss << endl;
    cout << "\tt Dimension of coordinates: \n" << Dim_xyz << endl;

    cout << "\tt Degrees of Freedom at Each Node: \n" << Node_DOF << endl;
    cout << "\tt Degrees of Freedom at Each Element: \n" << Ele_DOF << endl;
    cout << "\tt Total DOF of the Problem: \n" << Total_DOF << endl;
    cout << "\tt Gauss Point at Each Element: \n" << Num_ele_gauss << endl;
    cout << endl;
}
```
void GENERAL_DATA::assign_Node_DOF(int i)
{  
  Node_DOF = i;
  def_DOF();
}

void GENERAL_DATA::assign_Ele_type(int *in)
{  
  Num_ele_type= in[0];
  for(int i=0; i<Num_ele_type; i++)
    Ele_type[i]= in[i+1];
}

void GENERAL_DATA::def_DOF(void)
{  
  Ele_DOF= Num_ele_node*Node_DOF;
  Total_DOF= Num_nodal_point*Node_DOF;
}

void GENERAL_DATA::init_Node_freq(void)
{  
  int i;
  for(i=0; i<A_total_node; i++)
    Node_freq[i]= 0;
}
APPENDIX D

SOURCE CODE OF CONCURRENT OBJECT-ORIENTED COMPOSITE LAMINATE ANALYSIS

D.1 Source Code of GL_H

#ifndef GL_H
#define GL_H

/* Definition of classes for global-local model */

File name: gl.h 09/23/92

Coded by George G. Yu (The Ohio State University)

Compiled on <<AT&T C++ Translator 2.1.03 08/31/90>>

#include "general.c"
#include "vector.c"
#include "matrix.c"
#include "fem.c"
#include "my_sem.h"

// Classes declared in this head file
class GLOBAL_DATA;
class MATERIAL;
class GLOBAL_ELEMENT;
struct Standard_type;
class Object_Record;
class Version_Record;

// Constants for the global-local model
const A_num_mat= 5;
const A_num_layer= 10;
const A_num_node= 8;
const A_num_total_class= 20;
const A_num_total_object= 50;
const A_num_indiv_object= 30;
const A_num_version = 3;
const A_ele_share_node = 4;  // No. of ele sharing one node
const A_pipe_size = 4096;

struct ID
{
    int Num_ele_type;
    int shmid_GD;
    int shmid_vom;
    int Element[50];
};

class Object_Record
{
    public:
    Object_Record(void) {Type = NULL; Obj = 0; Act = NO; Ptr = -1;};
    ~Object_Record(void) {;};
    void assign_Obj(int i) {Obj = i;};
    void assign_Act(int i) {Act = i;};
    void assign_Type(int i) {Type = i;};
    void assign_Ptr(streampos i) {Ptr = i;};
    int get_Obj(void) const {return Obj;};
    int get_Act(void) const {return Act;};
    int get_Type(void) const {return Type;};
    streampos get_Ptr(void) const {return Ptr;};
    friend ostream& operator<<(ostream&, const Object_Record&);

    private:
    int Type;
    int Obj;
    int Act;
    streampos Ptr;
};

class Version_Record
{
friend ostream& operator<<(ostream&, const Version_Record&);

/*
int *T[5];    // 0: material, 1: node, 2: ele_seq, 3: B.C.
// 4: global_data
*/

short* get_T(int i) const {return T[i];};
int get_Ver(void) const {return Ver;};
int get_TO(int i) const {return T[i][0];};
char* get_Name(void) const {return Name;};

void Add_T(int i) {T[i][0]++;
void assign_Ver(int i) {Ver=i;};
void assign_Name(char* s) {strcpy(Name,s);};
void assign_T(int i1, int i2, int i3) {T[i1][i2][i3]= i3;};
private:
int Ver;
char Name[15];
short T[5][A_num_indiv_object];
};

class VOM
{//
public:
VOM(void);
~VOM(void) {};

void assign_VOM(Object_type, int, streampos);
void assign_Num_Ver(int i) {Num_Ver= i; VM[i].assign_Ver(i+1);};

int get_Num_OM(void) const {return Num_Obj;};
int get_Num_VM(void) const {return Num_Ver;};
Version_Record& get_VM(int i) const {return VM[i];};
Object_Record& get_OM(int i) const {return OM[i];};
short get_Object_Rec(Objects_type t) const {return Object_Rec[t][1];};

void Equal_VM(Objects_type, int, int);
streampos get_OM(Objects_type, int, int);

Bool Store(void);
Bool Retrieve(void);
friend ostream& operator<<(ostream&, const VOM&);
protected:
    void assign_VM(Object_type, int);
    void assign_OM(Object_type, streampos);

private:
    int Num_Ver;               // current working version
    int Num_Obj;
    Version_Record VM[A_num_version];
    Object_Record OM[A_num_total_object];
    short Object_Rec[A_num_total_class][2]; // class identifier and object number
};

class GLOBAL_DATA : public GENERAL_DATA
{
    public:
        GLOBAL_DATA(char*);
        GLOBAL_DATA();
        GLOBAL_DATA(const GLOBAL_DATA&);
        ~GLOBAL_DATA();

        void input_info_g(void);
        void check_info_g(void);

        int get_Num_total_layer() const {return Num_total_layer;} ;
        int get_Num_global_layer() const {return Num_global_layer;} ;
        int get_Num_material() const {return Num_material;} ;
        int get_Num_local_DOF() const {return Num_local_DOF;} ;
        int get_layer_type(int i) const {return Type[i-1];}
        int get_Mat_shm_id() const {return Mat_shm_id;} ;
        double get_total_height() const {return H;} ;
        double get_global_height() const {return H_global;} ;
        double get_layer_height(int i) const {return h[i-1];} ;

        void assign_global_layer(int);
        void assign_Mat_shm_id(int i) {Mat_shm_id= i;};
        void assign_Title(char* name) {strcpy(Title, name);} ;
        streampos Insert(char*, streampos);
        Bool Store(char*, streampos);
        Bool Retrieve(char*, streampos);

        streampos get_Offset(void) const {return Offset;} ;
        streampos get_Length(void) const {return Length;} ;
        Bool get_ACTIVE(void) const {return ACTIVE;} ;
}
private:
    // storage information
    Object_type Type_o;   // to distinguish the following Type
    Bool ACTIVE;
    streampos Offset;    // position head
    streampos Length;   // sizeof(GLOBAL_DATA)
    char File_name[15];

    int Num_total_layer;
    int Num_global_layer;
    int Num_material;
    int Num_global_DOF;
    int Num_local_DOF;
    int Mat_shm_id;
    double h[A_num_layer];
    int Type[A_num_layer];
    double H;
    double H_global;
};

///////////////////////////////
class MATERIAL
///////////////////////////////
{
    public:
        MATERIAL():S(6,6),C(6,6){};
        ~MATERIAL(){};

        void assign(VECTOR&);
        void assign(double *);
        MATRIX& get_S(void) {return S;};
        MATRIX& get_C(void) {return C;};
        void def_S(void);
        void def_C(void);

        friend ostream& operator<<(ostream&, const MATERIAL&);

    protected:
        void def_Trss(MATRIX&);
        void def_Tran(MATRIX&);

    private:
        double m,n,mm,nn,mn;

        MATRIX S;
        MATRIX C;
class GLOBAL_ELEMENT :public ELEMENT

public:
  GLOBAL_ELEMENT(int, GLOBAL_DATA&, MATERIAL&, VOM&);
  ~GLOBAL_ELEMENT();

  void definition();

private:
  double H_global;
  int Node_DOF;
  int Num_global_DOF;
  int Ele_Type;
  int Num_Total_Layer;
  int Num_Global_Layer;
  int Num_Material;

  MATRIX C;
};

class LOCAL_ELEMENT :public ELEMENT

public:
  LOCAL_ELEMENT(int, GLOBAL_DATA&, MATERIAL&, int, VOM&);
  ~LOCAL_ELEMENT();

  void definition();

private:
  double H;
  int Layer_num;
  int Node_DOF;
  int Num_layer_DOF;
  int Ele_Type;
  int Num_Material;

  MATRIX S;
};
struct Standard_type
{
    int Num;
    int Num_node;
    double Edge_ratio[6];
};

class ELEMENT_TYPE
{
    public:
    ELEMENT_TYPE(GLOBAL_DATA&);
    ~ELEMENT_TYPE();
    int *def_ele_type(VOM&, int);

    private:
    int *Ele_type;
    int Num_element;
    int Num_node;
    int Num_edge;
    int Num_total_point;
    int Num_dim;
    int Num_type;
    int Num_DOF;

    Standard_type st[A_ele_type];
    int Add_type(Standard_type &);
};

#ifend
D.2 Source Code of GL.C

#include "gl.h"

// Declaring subroutines in FORTRAN
extern "C"
{
extern void lxxn_ (double *a12n, double *a12bn, double *a34n,
 double *a3bn, double *a56n, double *shape,
 double *cartd, int *mdim, int *nnode, double *hglob);
extern void vl_ (double *v, double *ak, double *h, double *S);
extern void ssl_1_ (double *s11, double *ak, double *v);
extern void ss22_ (double *s22, double *ak, double *v);
extern void rl_1_ (double *rl, double *v, double *shape, double *cartd,
 int *nnode);
extern void r22_ (double *r2, double *v, double *ak, double *shape,
 double *cartd, int *nnode, double *h);
}

/* init_queue(key_t KEY) */
int init_queue(key_t KEY)
/* */
{
int queue_id;
if((queue_id= msgget(KEY, IPC_CREAT|QPERM))==-1)
 perror("msgget failed");
 return (queue_id);
}
int remove_queue(int q_id)
{
    if(msgctl(q_id, IPC_RMID, (struct msqid_ds*)0) < 0)
    {
        perror("msgctl failed in destroying queue");
        exit(1);
    }

    return (0);
}

int Queue_report(int queue_id, int sw)
{
    struct msqid_ds msq_status;
    if(msgctl(queue_id, IPC_STAT, &msq_status) < 0)
    {
        perror("msgctl failed in reading");
        exit(1);
    }

    if(sw.ON) // output info on screen
    {
        cout<<"Information about Queue:"<<endl;
        cout<<"Message(s) on queue"<<endl;
        cout<<"Bytes on the queue"<<endl;
        cout<<"Last send by proc"<<endl;
        cout<<"Last recv by proc"<<endl;
    }

    return (msq_status.msg_qnum);
}

ostream& operator<<(ostream& s, const Object_Record& m)
{
    s<<"Object type is "<<Object_string[m.get_Type()]<<endl;
    s<<"Object number is "<<m.get_Obj()<<endl;
    if(m.get_Act()==YES)
        s<<"Record is ACTIVE on the file."<<endl;

else
    s<<"Record is INACTIVE on the file."<<endl;
    s<<"The head of the record is at ":m.get_Ptr()<<endl;
    s<<endl;

    return s;
}

/*****************************/
Version_Record::Version_Record(void)
/*****************************/
{
    /*
        *{ T[0]= new int[3];
        T[1]= new int[30];
        T[2]= new int[30];
        T[3]= new int[3];
        */
        int i, j;
        for(i=0; i<5; i++)
            for(j=0; j<A_num_indiv_object; j++)
                T[i][j]= 0;

        Ver= 0;
    }

/*****************************/
Version_Record::~Version_Record(void)
/*****************************/
{
    /*
        *{ int i;
        for(i=0; i<4; i++)
            if(T[i]!=NULL)
                delete [] T[i];
        */
    }

/*****************************/
ostream& operator<< (ostream& s, const Version_Record& v)
/*****************************/
{
    int j,k;
    short* T;
    s<<"\n\nVersion= "<<v.get_Ver()<<endl;
    for(j=0; j<5; j++)
    {
        T= v.get_T(j);
        s<<"Class= "<<j+1<<endl;
    }
if(T[0]== 0)
    s<<"No object";
else
{
    s<<T[0]<<" objects: ";
    for(k=1; k<=T[0]; k++)
        s<<T[k]<<" ";
}
    s<<endl;
}
s<<endl;
return s;

/*****************************/
VOM::VOM(void)
/*****************************/
{
    Num_Obj= 0;
    Num_Ver= 0;

    for(int i=0; i<20; i++)
        Object_Rec[i][i]= 0;

    Object_Rec[0][0]= NOTHING;
    Object_Rec[1][0]= OVECTOR;
    Object_Rec[2][0]= OMATRIX;
    Object_Rec[3][0]= OGENERAL_DATA;
    Object_Rec[4][0]= OSHAPE;
    Object_Rec[5][0]= OSHAPE_4;
    Object_Rec[6][0]= OSHAPE_6;
    Object_Rec[7][0]= OJACOB;
    Object_Rec[8][0]= ONODE;
    Object_Rec[9][0]= OMATERIAL;
    Object_Rec[10][0]= OGAUSS;
    Object_Rec[11][0]= OELEMENT;
    Object_Rec[12][0]= OELEMENT_SEQ;
    Object_Rec[13][0]= OBOUCOND;
    Object_Rec[14][0]= OGLOBAL_DATA;
    Object_Rec[15][0]= OGLOBAL_ELEMENT;
    Object_Rec[16][0]= OLOCAL_ELEMENT;
    Object_Rec[17][0]= OELEMENT_TYPE;
}

/*****************************/
void VOM::assign_VOM(Object_type Type, int Obj, streampos pos)
/*****************************/
{
    Num_Obj++;
}
if(Num_Obj>=A_num_total_object)
{
    cout<<"WARNING: Object number of one class is "<<Num_Obj
    <<", over limitation of "<<A_num_total_object<<endl;
    exit(-1);
}
else
{
    Object_Rec[Type][1]++;
    if(Object_Rec[Type][1]< A_num_indiv_object)
    {
        assign_OM(Type, pos);
        if(Type==OMATERIAL || Type==ONODE || Type==OELEMENT_SEQ ||
            Type==OBUCOND || Type==OGLOBAL_DATA)
            assign_VM(Type, Obj);
    }
    else
    {
        cout<<"WARNING: Object number is "<<Object_Rec[Type][1]<<
            ", over limitation of "<<A_num_indiv_object<<endl;
        exit(-1);
    }
}

//**********************************************************************
void VOM::assign_OM(Object_type Type, streampos pos)
//**********************************************************************
{
    OM[Num_Obj-1].assign_Type(Type);
    OM[Num_Obj-1].assign_Obj(Object_Rec[Type][1]);
    OM[Num_Obj-1].assign_Act(YES);
    OM[Num_Obj-1].assign_Ptr(pos);
}

//**********************************************************************
void VOM::assign_VM(Object_type Type, int Obj)
//**********************************************************************
{
    int which;
    if(Type==OMATERIAL)
    which=0;

else if (Type == ONODE)
    which = 1;
else if (Type == OELEMENT_SEQ)
    which = 2;
else if (Type == OBOUCOND)
    which = 3;
else if (Type == OGLOBAL_DATA)
    which = 4;

VM[Num_Ver].Add_T(which);
VM[Num_Ver].assign_T(which, Obj, Object_Rec[Type][1]);

void VOM::Equal_VM(Object_type Type, int v1, int v2)
{
    int which, j;
    if (Type == OMATERIAL)
        which = 0;
    else if (Type == ONODE)
        which = 1;
    else if (Type == OELEMENT_SEQ)
        which = 2;
    else if (Type == OBOUCOND)
        which = 3;
    else if (Type == OGLOBAL_DATA)
        which = 4;

    for (j = 0; j <= VM[v1].get_T0(which); j++)
        VM[v2].assign_T(which, j, (VM[v1].get_T(which))[j]);
}

streampos VOM::get_OM(Object_type Type, int Obj, int Ver)
{
    int i;
    int temp_Obj;

    if (Type == OMATERIAL)
        temp_Obj = (VM[Ver].get_T0()[Obj]);
    else if (Type == ONODE)
        temp_Obj = (VM[Ver].get_T1()[Obj]);
    else if (Type == OELEMENT_SEQ)
        temp_Obj = (VM[Ver].get_T2()[Obj]);
    else if (Type == OBOUCOND)
        temp_Obj = (VM[Ver].get_T3())[Obj];
else if (Type == OGLOBAL_DATA)
    temp_Obj = (VM[Ver].get_T(4))[Obj];
else
    temp_Obj = Obj;

for (i = 0; i < Num_Obj; i++)
{
    if ((OM[i].get_Type() == Type) && (OM[i].get_Obj() == temp_Obj) &&
        (OM[i].get_Act() == YES))
        return OM[i].get_Ptr();
}
return -1;

/**/ 
Bool VOM::Store(void)
/**/
/**/ 
{
    ofstream fout;
    fout.open("vom.dat",ios::in|ios::out);
    if (!fout)
    {
        cout << "Cannot open file <vom.dat>" << endl;
        return FALSE;
    }
    else
    {
        fout.seekp(0, ios::beg);
        fout.write((char*)&(This), sizeof(VOM));
        fout.close();
        return TRUE;
    }
}
/**/ 
Bool VOM::Retrieve(void)
/**/
/**/ 
{
    ifstream fin;
    fin.open("vom.dat",ios::in|ios::out);
    if (!fin)
    {
        cout << "Cannot open file <vom.dat>" << endl;
        return FALSE;
    }
    else
    {
```cpp
fin.seekg(0, ios::beg);
fin.read((char*)&(This), sizeof(VOM));
fin.close();
return TRUE;
}

/*****************************/
ostream& operator<<(ostream& s, const VOM& v)
/*****************************/
{
int i;
// Version_Record VM;
// Object_Record OM;

s<<"\n\tOutput of Version Manager Information\n"<<endl;

s<<"\tVersion number: "<<v.get_Num_VM()+1<<endl;

s<<"\tObject number: "<<v.get_Num_OM()<<endl;

s<<"\n\tOutput of OM data\n"<<endl;

for(i=0; i<v.get_Num_OM(); i++)
{
  s<<"\n\tObject #= "<<i+1<<endl;
  s<<v.get_OM(i);
}

s<<"\n\tOutput of first VM data\n"<<endl;

// for(i=0; i<v.get_Num_VM(); i++)
for(i=0; i<1; i++)
{
  // s<<"\n\tVersion= "<<i+1<<endl;
  s<<v.get_VM(i);
}

s<<endl;
return s;
}

/*****************************/
GLOBAL_DATA::GLOBAL_DATA(char *name)
/*****************************/
{
strcpy(Title, name);
}

/*****************************/
GLOBAL_DATA::GLOBAL_DATA(const GLOBAL_DATA& S)
```
/**
 * Input information
 */

#include "GLOBAL_DATA.h"

GLOBAL_DATA::GLOBAL_DATA()
{
    Num_nodal_point = S.Num_nodal_point;
    Num_element = S.Num_element;
    Num_bc_point = S.Num_bc_point;
    Num_total_layer = S.Num_total_layer;
    Num_global_layer = S.Num_global_layer;
    Num_material = S.Num_material;
    Num_ele_node = S.Num_ele_node;
    Num_gauss = S.Num_gauss;
    Node_DOF = S.Node_DOF;
    Total_DOF = S.Total_DOF;
    Ele_DOF = S.Ele_DOF;
    Num_ele_gauss = S.Num_ele_gauss;
} /*
 * Global data
 */

for(int i=0; i<Num_Obj; i++)
{
    OM[i].Type = S.OM[i].Type;
    OM[i].Obj = S.OM[i].Obj;
    OM[i].Act = S.OM[i].Act;
    OM[i].Ptr = S.OM[i].Ptr;
}
/*
 * Copy title
 */

strcpy(Title, S.Title);

/**
 * Input information
 */

void GLOBAL_DATA::input_info_g()
{
    int i;

    cout << "Please input the following additional information.\n" << endl;
    cout << "Number of Material Types: " ;
    cin >> Num_material;
    if(Num_material > A_num_mat)
    {
        cout << "Only " << A_num_mat << " material types are allowed!!" << endl;
        cout << "Please check and re-enter: " ;
        cin >> Num_material;
    }

    cout << "Total Number of Layers: " ;
    cin >> Num_total_layer;
    if(Num_total_layer > A_num_layer)
    {
        cout << "Only " << A_num_layer << " layers are allowed!!" << endl;
        cout << "Please check and re-enter: " ;
    }
}
cin >> Num_total_layer;
else
    
    H = 0.0;
    cout << "Please input each ply's material type and height:" << endl;
    for (i = 0; i < Num_total_layer; i++)
    {
        cout << "Layer " << i + 1 << " : ";
        cin >> Type[i];
        cin >> h[i];
        H += h[i];
    }
}

cout << "Total Number of Global Layers: " << endl;
cin >> Num_global_layer;
assign_global_layer(Num_global_layer);

/*********************/
void GLOBAL_DATA::check_info_g(void)
/*********************/
{
    int i;

    VECTOR mat_prop(10);
    MATERIAL Mat;

    cout << "Review the Additional Information Provided.\n" << endl;
    cout << "Total Number of Layers: " << Num_total_layer << endl;
    cout << "Total Number of Global Layers: " << Num_global_layer << endl;
    cout << "Number of Types of Materials: " << Num_material << endl << endl;

    cout << "Layer\nMat. Type\nHeight" << endl;
    for (i = 0; i < Num_total_layer; i++)
    {
        cout << "\t" << i + 1 << " \t" << Type[i] << "\t" << h[i] << " (in)" << endl;
    }
    cout << endl;
}

/*********************/
void GLOBAL_DATA::assign_global_layer(int i)
/*********************/
{
    int j;
Num_global_layer = i;
H_global = 0.0;
for (j = 0; j < Num_global_layer; j++)
    H_global += h[j];

Node_DOF = 13 + (Num_total_layer - Num_global_layer) * 7;
Num_global_DOF = 13 * Num_ele_node;
Num_local_DOF = ((Num_total_layer - Num_global_layer) * 7) * Num_ele_node;
assign_Node_DOF(Node_DOF);

streampos GLOBAL_DATA::Insert(char* f_name, streampos Start)
{
    fstream f;
    streampos New_Start, End;
    int size_gd = sizeof(GLOBAL_DATA);
    GLOBAL_DATA temp;
    int input;

    f.open(f_name, ios::in | ios::out);
    if(!f)
    {
        cout << "Cannot open file " << f_name << endl;
        return -1;
    }
    else
    {
        f.seekg(0, ios::end);
        End = f.tellg();
        if(Start != -1)
        {
            if(End > Start) // avoiding out of file
            {
                f.seekg(Start, ios::beg);
                f.read((char*) &temp, size_gd);
                if(temp.getLength() < size_gd)
                    New_Start = End;
                else
                {
                    if(temp.getActive() == YES)
                    {
                        cout << "Are you sure to override the existing record?"
                        << "(0/1) (NO/YES)" << endl;
                        cin >> input;
                        if(input == NO)
                            New_Start = End;
                    }
                }
            }
        }
    }
}
else
    New_Start= Start;
}
else
    New_Start= Start;
}
else
    New_Start= End;
f.close();

if(This.Store(f_name, New_Start)== TRUE)
    return New_Start;
else
    return -1;
}

bool GLOBAL_DATA::Store(char* f_name, streampos Start)
{/**************************************************/
  
  
  fstream f;
  This.ACTIVE= TRUE;
  This.Length= sizeof(GLOBAL_DATA);

  strcpy(File_name,f_name);
  f.open(File_name,ios::in|ios::out);
  if(!f)
  {
    cout<<"Cannot open file "<<File_name<<endl;
    return FALSE;
  }
  else
  {
    f.seekp(Start, ios::beg);
    f.write((char*)&(This), sizeof(GLOBAL_DATA));
    f.close();
    return TRUE;
  }

  }/**************************************************/

bool GLOBAL_DATA::Retrieve(char* File_name, streampos Start)
{/**************************************************/
  

fstream f;

f.open(File_name.ios::in|ios::out);
if(!f)
{
    cout<<"Cannot open file "<<File_name<<endl;
    return FALSE;
}
else
{
    f.seekg(Start, ios::beg);
    f.read((char*)&(This), sizeof(GLOBAL_DATA));
    f.close();
    return TRUE;
}

/**************************************************************************/
void MATERIAL::assign(VECTOR& props)
/**************************************************************************/
{
    E11= props[0]; E22= props[1]; E33= props[2];
    G12= props[3]; G23= props[4]; G13= props[5];
    v12= props[6]; v23= props[7]; v13= props[8];
    v21= v12*E22/E11; v32= v23*E33/E22; v31= v13*E33/E11;
    angle= props[9]*3.14159/180.;
}

/**************************************************************************/
void MATERIAL::assign(double *props)
/**************************************************************************/
{
    E11= props[0]; E22= props[1]; E33= props[2];
    G12= props[3]; G23= props[4]; G13= props[5];
    v12= props[6]; v23= props[7]; v13= props[8];
    v21= v12*E22/E11; v32= v23*E33/E22; v31= v13*E33/E11;
    angle= props[9]*3.14159/180.;
}

void MATERIAL::def_Trss(MATRIX& Trss)
/**************************************************************************/
{
    m= cos(angle);
    n= sin(angle);
    mm= m*m;
    nn= n*n;
    mn= m*n;
    Trss[0][0]= mm; Trss[0][1]= nn; Trss[0][5]= 2.*mn;
    Trss[1][0]= nn; Trss[1][1]= mm; Trss[1][5]= -2.*mn;
void MATERIAL::def_Tran(MATRIX& Tran)
{  // def_Tran(Tran)
    m = cos(angle);
    n = sin(angle);
    mm = m*m;
    nn = n*n;
    mn = m*n;
    Tran[0][0] = mm;  Tran[0][1] = nn;  Tran[0][5] = mn;
    Tran[1][0] = nn;  Tran[1][1] = mm;  Tran[1][5] = -mn;
    Tran[2][2] = 1.;  Tran[3][3] = m;  Tran[3][4] = -n;
    Tran[4][3] = n;  Tran[4][4] = m;  Tran[5][0] = -2.*mn;
    Tran[5][1] = 2.*mn;  Tran[5][5] = mm-nn;
}

void MATERIAL::def_S(void)
{  // def_S()
    MATRIX Tran(6,6);
    S[0][0] = 1./E11;  S[0][1] = -v12/E11;
    S[0][2] = -v13/E11;  S[1][0] = -v21/E22;
    S[1][1] = 1./E22;  S[1][2] = -v23/E22;
    S[2][0] = -v31/E33;  S[2][1] = -v32/E33;
    S[2][2] = 1./E33;  S[3][3] = 1./G23;
    S[4][4] = 1./G13;  S[5][5] = 1./G12;
    if(angle!= 0.0)
    {
        def_Tran(Tran);
        MATRIX TranT(6,6);
        TranT = trans(Tran);
        Tran *= S;
        Tran *= TranT;
        S = Tran;
    }
}

void MATERIAL::def_C(void)
{  // def_C()

MATRX Trss(6,6):
doub1e SS:

SS= 1.0-v12*v21-v23*v32-v31*v13-2.0*v21*v32*v13;
C[0][0]= El1*(1.0-v23*v32)/SS;
C[0][1]= El1*(v21+v31*v23)/SS;
C[0][2]= El1*(v31+v21*v32)/SS;
C[1][0]= E22*(v12+v32*v13)/SS;
C[1][1]= E22*(-1.0-v31*v13)/SS;
C[1][2]= E22*(v32+v12*v31)/SS;
C[2][0]= E33*(v13+v12*v23)/SS;
C[2][1]= E33*(v23+v21*v13)/SS;
C[2][2]= E33*(-1.0-v12*v21)/SS;
C[3][3]= G23;
C[4][4]= G13;
C[5][5]= G12;

if(angle!= 0.0)
{
    def_Trss(Trss);
    MATRX TrssT= trans(Trss);
    Trss *= C;
    Trss *=TrssT;
    C= Trss;
}

/*==============================================*/
ostream& operator<<(ostream& s, const MATERIAL &m)
/*==============================================*/
{
    s<"\tE11= "<<m.E11<<" Psi;\tE22= "<<m.E22<<" Psi;"<<endl;
    s<"\tE33= "<<m.E33<<" Psi;\tG12= "<<m.G12<<" Psi;"<<endl;
    s<"\tG23= "<<m.G23<<" Psi;\tG13= "<<m.G13<<" Psi;"<<endl;
    s<"\tv12= "<<m.v12<<";\tv23= "<<m.v23<<";"<<endl;
    s<"\tv13= "<<m.v13<<";\tttheta= "<<m.angle*180/3.14159<<";"<<endl;
    s< "endl;
    return s;
}

/*==============================================*/
GLOBAL_ELEM
EN*/

GLOBAL_ELEMENT::GLOBAL_ELEMENT(int num, GLOBAL_DATA& GD,
    MATERIAL& Mat, VOM& vom)
    :ELEMENT(num,
GD.get_Num_global_DOF(),
GD.get_Node_DOF(),
GD.get_Dim_xyz(),
GD.get_Num_ele_node(),
GD.get_Num_nodal_point(),
GD.get_Num_gauss(),
GD.get_Num_element()),

C(6,6)

/*

Mat.def_C();
C= Mat.get_C();
H_global= GD.get_global_height();
Num_global_DOF= GD.get_Num_global_DOF();

int i;
streampos *Pos1, *Pos2;
Pos1= new streampos[GD.get_Num_nodal_point()];
Pos2= new streampos[GD.get_Num_element()];

int Ver= vom.get_Num_VM();
for(i=0; i< GD.get_Num_nodal_point(); i++)
    Pos1[i]= vom.get_OM(ONODE,i+1,Ver);
for(i=0; i< GD.get_Num_element(); i++)
    Pos2[i]= vom.get_OM(OELEMENT_SEQ,i+1,Ver);

def_Inods(Pos2);
def_Elcod(Pos1);

delete [] Pos1;
delete [] Pos2;
*/

cout<<"lnods:"<< "<< (get_Inods())[0]<< "
    "
    "<< (get_Inods())[1]<< "
    "
    "<< (get_Inods())[2]<< "
    "
    "<< (get_Inods())[3]<< endl;
*/
}
SHAPE_4 Shape:
/* if(Num_N0de== 4)
{
 SHAPE_4 Shape;
}
else
{
 SHAPE_6 Shape;
}
*/
MATRIX CXX(4,4), CYX(2,4), CYY(2,2);  
MATRIX A1(4,Num_DOF), A1T(Num_DOF,4), A2(2,Num_DOF),
A2T(Num_DOF,2);
fshape= new double[Num_N0de];
fcartd= new double[Dim_xyz*Num_N0de];
fa12n= new double[13*Num_N0de*4];
fa12bn= new double[13*Num_N0de*2];
fa34n= new double[13*Num_N0de*4];
fa3bn= new double[13*Num_N0de*2];
fa56n= new double[13*Num_N0de*2];

for(i= 0; i< 3; i++)
{
 CXX[i][3]= C[i][5];
 CXX[3][i]= C[5][i];
 CYX[0][i]= C[i][3];
 CYX[1][i]= C[i][4];
 for(j=0; j<3; j++)
  CXX[i][j]= C[i][j];
}
 CXX[3][3]= C[5][5];
 CYX[0][3]= C[5][3];
 CYX[1][3]= C[5][4];
 CYY[0][0]= C[3][3];
 CYY[0][1]= C[3][4];
 CYY[1][0]= C[3][4];
 CYY[1][1]= C[4][4];

//cout<<"\nComputing Global region. Element #"<<get_Ele_Num()<<endl;

for (i=0; i<Num_Gauss; i++)
// for (i=0; i<1; i++)
{
 s= Position[i];
 for (j=0; j<Num_Gauss; j++)
// for (j=0; j<1; j++)
{
 t= Position[j];

k++;
// cout<<"Gauss point "<<k<<endl;
Shape.definition(s,t);
(Shape.get_shape()).VtoF(fshape,4);
Jacob.definition(Shape.get_derive());
// cout<< "In djacb= "<< Jacob.get_djacb()<< endl;
Cartd= Jacob.get_cartd();
Cartd.MtoF(fcartd,2,4);
Dvolu= Jacob.get_djacb()*Weight[i]*Weight[j];
// cout<< "cartd:"<<Cartd;
// cout<<"Dvolu= "<<Dvolu<<endl;

c1= H_global*Dvolu;
c2= c1;
c8= H_global*H_global*c1/12.0;
c9= c8;
c10= c8;
c12= c1*H_global*H_global*H_global*H_global/320.0;

int i2= 4;
lxxn_(fa12n, fa12bn, fa34n, fa3bn, fa56n, fshape, fcartd, &Num_global_DOF, &i2, &H_global);
A1.FtoM(fa12n,4,Num_global_DOF);
A1T= trans(A1);
Stiff+= c1*A1T*CXX*A1;
A2.FtoM(fa12bn,2,Num_global_DOF);
A2T= trans(A2);
Stiff+= c2*A2T*CYY*A2;
A1.FtoM(fa34n,4,Num_global_DOF);
A1T= trans(A1);
Stiff= c8*A1T*CXX*A1;
A2.FtoM(fa3bn,2,Num_global_DOF);
A2T= trans(A2);
Stiff+= c9*A2T*CYY*A2;
A2.FtoM(fa56n,2,Num_global_DOF);
A2T= trans(A2);
Stiff+= c10*A2T*CYY*A2;
A2.FtoM(fa56n,2,Num_global_DOF);
A2T= trans(A2);
Stiff+= c12*A2T*CYY*A2;

}
```cpp
// cout<<" Output of Stiff[0][0]= "<<Stiff[0][0]<<endl;

delete [13*Num_Node*4] fa12n;
delete [13*Num_Node*2] fa12bn;
delete [13*Num_Node*4] fa34n;
delete [13*Num_Node*2] fa3bn;
delete [13*Num_Node*2] fa56n;
delete [Num_Node] fshape;
delete [Dim_xyz*Num_Node] fcartd;
}

/local_element::LOCAL_ELEMENT(int num, GLOBAL_DATA& GD,
   MATERIAL& Mat, int L_num, VOM& vom)
  :ELEMENT(num,
   GD.get_Num_global_DOF(),
   GD.get_Node_DOF(),
   GD.get_Dim_xyz(),
   GD.get_Num_ele_node(),
   GD.get_Num_nodal_point(),
   GD.get_Num_gauss(),
   GD.get_Num_element()).
S(6,6)
{
  Mat.def_S();
  S= Mat.get_S();
  Layer_num= L_num;
  H= GD.get_layer_height(Layer_num);
  Num_layer_DOF= 13*Num_Node;

  int i;
  streampos *Posl, *Pos2;
  Posl= new streampos[G.D.get_Num_nodal_point()];
  Pos2= new streampos[G.D.get_Num_element()];

  int Ver= vom.get_Num_VM();
  for(i=0; i< GD.get_Num_nodal_point(); i++)
    Posl[i]= vom.get_OM(ONODE,i+1,Ver);
  for(i=0; i< GD.get_Num_element(); i++)
    Pos2[i]= vom.get_OM(OELEMENT_SEQ,i+1,Ver);
```
def_lnods(Pos2):
def_Elcod(Pos1);

delete [] Pos1;
delete [] Pos2;
/
   cout<<"Ele_Num:"<<get_Ele_Num()<<endl;
   cout<<"lnods:"" "<<(get_lnods())[0]"" 
     <<(get_lnods())[1]"" 
     <<(get_lnods())[2]"" 
     <<(get_lnods())[3]<<endl;
/*
void LOCAL_ELEMENT::definition(void)
/
{
    int i,j,k;
    double s,t;
    k=0;
    JACOB Jacob(Num_Node,Elcod);
    // if(Nuin_Node= 4)
    // {
    //     SHAPE_4 Shape;
    // }
    // else
    // {
    //     SHAPE_6 Shape;
    // }
    MATRIX R1(10,Num_layer_DOF), R2(6,Num_layer_DOF), 
    R1T(Num_layer_DOF,10),R2T(Num_layer_DOF,6), 
    S11(10,10), S22(6,6);

    fshape= new double[Num_Node];
    fcartd= new double[Dim_xyz*Num_Node];
    fak= new double[23];
    fS= new double[6*6];
    fv= new double[10*10];
    fs11= new double[10*10];
    fs22= new double[6*6];
    fr1= new double[10*Num_layer_DOF];
    fr2= new double[6*Num_layer_DOF];
    S.MtoF(fS,6,6);
vl_ (fv,fak,&H,fS);
ss11_(fs11,fak,fv);
ss22_ (fs22,fak,fv);
S11.FtoM(fs11,10,10);
S22.FtoM(fs22,6,6);

// cout<<"lnComputing Local region. Element ";<<get_Ele_Num()<<
// ";Layer ";<<Layer_num+1<<endl;

// for (i=0; i<Num_Gauss; i++)
for (i=0; i<1; i++)
{
    s = Position[i];
    // for (j=0; j<Num_Gauss; j++)
    for (j=0; j<1; j++)
    {
        t = Position[j];
        k++;
        // cout<<"Gauss point ";<<k<<endl;
        Shape.definition(s,t);
        (Shape.get_shape()).VtoF(fshape,4);
        Jacob.definition(Shape.get_derive());
        // cout<<"ln djacb= "<< Jacob.get_djacb()<<endl;
        Cart= Jacob.get_cartd();
        Cartd.MtoF(fcartd,2,4);
        Dvolu= Jacob.get_djacb()*Weight[i]*Weight[j];
        // cout<<"cartd: "<<Cartd;
        // cout<<"Dvolu= "<<Dvolu<<endl;

        r11_ (fr1.fv, fshape, fcartd, &Num_Node);
        R1.FtoM(fr1,10,Num_layer_DOF);
        R1T= trans(R1);
        Stiff += Dvolu*R1T*S11*R1;

        r22_ (fr2.fv,fak.fshape,fcartd,&Num_Node,&H);
        R2.FtoM(fr1,6,Num_layer_DOF);
        R2T= trans(R2);
        Stiff += Dvolu*R2T*S22*R2;
    }
}

// cout<<" Output of Stiff[0][0]= "<<Stiff[0][0]<<endl<<endl;

delete [Num_Node] fshape;
delete [Dim_xyz*Num_Node] fcartd;
delete [23] fak;
delete [6*6] fS;
delete [10*10] fv;
delete [10*10] fs11;
delete [6*6] fs22;
delete [10*Num_layer_DOF] fr1;
delete [6*Num_layer_DOF] fr2;

/*****************************/
ELEMENT_TYPE::ELEMENT_TYPE(GLOBAL_DATA& GD)
/*****************************/
{
    Ele_type= new int[GD.get_Num_element()];
    Num_type= 0;
    Num_element= GD.get_Num_element();
    Num_node= GD.get_Num_ele_node();
    Num_edge= GD.get_Num_ele_node(); // This info. should be included in GD!!
    Num_total_point= GD.get_Num_nodal_point();
    Num_dim= GD.get_Dim_xyz();
    Num_DOF= GD.get_Node_DOF();
}

/*****************************/
ELEMENT_TYPE::~ELEMENT_TYPE()
/*****************************/
{
    if(Ele_type!= NULL)
        delete [Num_element] Ele_type;
}

/*********************/
int* ELEMENT_TYPE::def_ele_type(VOM& vom, int Ver)
/*********************/
{
    int i, j, k;
    int Ele_type_info[A_ele_type+1];
    int *Lnods;
    double Elcod[3][A_num_node];
    double Edgel, Edge2;
    Standard_type s;
    ELEMENT_SEQ e;
    NODE node;

    Lnods= new int(A_num_node);
    streampos size_ele= sizeof(ELEMENT_SEQ);
    streampos Node_length= sizeof(NODE)+(Num_dim+Num_DOF)*sizeof(double) 
        +sizeof(VECTOR)*2;
// test of queue 7/10/92

// send Lnod information to queue
int queue_id1, queue_id2;

queue_id1 = init_queue(SEQ_QKEY);
queue_id2 = init_queue(NODE_QKEY);

struct q_jentry out_entry1; // nodal seq info of each ele
struct q_dentry out_entry2; // nodal coord

int num = Queue_report(queue_id1, OFF);

for (i=0; i<Num_element; i++)
{ // to get file info of e.
  e.Retrieve("try.dat", vom.get_OM(OELEMENT_SEQ, i+1, Ver));

  if(msgrcv(queue_id1, (msgbuf*)&out_entry1, MAXOBN,
           (long)(i+1), MSG_NOERROR)==-1)
  {
    perror("msgrcv failed");
    exit(1);
  }
  else
  {
    for(j=0; j<Num_node; j++)
    {
      // Coord. of each node
      if(msgrcv(queue_id2, (msgbuf*)&out_entry2, MAXOBN, 
           (long)out_entry1.mtext[j], MSG_NOERROR)==-1)
      {
        perror("msgrcv failed");
        exit(1);
      }
      else
      {
        for(k=0; k<Num_dim; k++)
          Elcod[k][j] = out_entry2.mtext[k];
      }
    }
  }
}

Edge1 = sqrt((Elcod[0][0]-Elcod[0][Num_node-1])^2
              + (Elcod[1][0]-Elcod[1][Num_node-1])^2
              + (Elcod[2][0]-Elcod[2][Num_node-1])^2);

s.Edge_ratio[0] = 1.0;
for(k= 0; k<Num_node-1; k++)
{  Edge2 = sqrt((Elcod[0][k]-Elcod[0][k+1])^2
                 + (Elcod[1][k]-Elcod[1][k+1])^2
                 + (Elcod[2][k]-Elcod[2][k+1])^2);
(Ecod[l][k]-Ecod[l][k+1]);

    s.Edge_ratio[k+1] = Edge2/Edge1;

    }
    s.Num_node = Num_node;
    if(Add_type(s) == YES)
        Ele_type_info[Num_type] = i+1;
        e.assign_Type(Num_type);
        e.Update(Num_type);
    }

if(remove_queue(queue_id1) != 0)
    cout << "Unsuccessful in removing ndoe seq queue" << endl;
if(remove_queue(queue_id2) != 0)
    cout << "Unsuccessful in removing ndoe queue" << endl;

Ele_type_info[0] = Num_type;

delete Lnodes;

return Ele_type_info;


/**********************************************************/
int ELEMENTTYPE::Add_type(Standard_type &s)
/**********************************************************/
{
    int i,j;
    int Add_type;

    Add_type = YES;

    for (i=0; i<Num_type; i++)
    {
        if(st[i].Num_node == s.Num_node)
        {
            Add_type = NO;
            for(j=0; j<Num_node; j++)
                if( st[i].Edge_ratio[j] != s.Edge_ratio[j] )
                {
                    Add_type = YES;
                    j = Num_node;
                }
        }
    }

    if(Add_type == YES)
    {
        Num_type ++;
        st[Num_type-1].Num = Num_type;
        st[Num_type-1].Num_node = s.Num_node;
    }
for(i=0; i< Num_node; i++)
    st[Num_type-1].Edge_ratio[i] = s.Edge_ratio[i];
}
return Add_type;

BOOL Object_detector(char* f_name, streampos Start)
{
  fstream f;
  Object_type Type;
  BOOL ACTIVE;
  streampos Offset;
  streampos Length;
  streampos File_length;
  streampos pos, End;
  int count;

  f.open(f_name, ios::in);
  if(!f)
  {
    cout << "Cannot open file " << f_name << endl;
    return FALSE;
  }
  else
  {
    f.seekg(0, ios::end);
    File_length = f.tellg();
    cout << "Output from Object Detector on File" << endl;
    cout << "The file " << f_name << "'s size is " << File_length << endl;
    pos = Start;
    count = 0;
    while(Start < File_length)
    {
      f.seekg(pos, ios::beg);
      f.read((char*)&Type, 4);
      f.read((char*)&ACTIVE, 4);
      f.read((char*)&Offset, 4);
      f.read((char*)&Length, 4);

      End = Length + Offset;
      cout << "The object " << count + 1 << " on the file is instance of " <<
          "class " << Object_string[Type] << endl;
      cout << "It starts at " << Start << " and end at " << End << endl;

      Start = End;
      pos = End;
    }
  }

count++; } return TRUE; }
#endif
D.3 Source Code of MAIN.C

/* Global-Local Model —--- MAIN Module */

File name: main.c 10/09/92

Coded by George G. Yu (The Ohio State University)

Compiled by <<AT&T C++ Translator 2.1.03 08/31/90>>

Concurrency is simulated with UNIX (SUNOS 4.1.1) system calls

A list of channels used for synchronization and IPC

<table>
<thead>
<tr>
<th>channel</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch1</td>
<td>sending shared memory identifier of GD</td>
</tr>
<tr>
<td>ch2</td>
<td>sending shared memory identifier of vom</td>
</tr>
<tr>
<td>ch3</td>
<td>synchronizing modules &quot;input&quot; and &quot;main&quot;</td>
</tr>
<tr>
<td>ch4</td>
<td>sending object of struct ID (defined in gl.h)</td>
</tr>
<tr>
<td></td>
<td>main-&gt;stiffness stiffness-&gt;-&gt;-&gt;element</td>
</tr>
<tr>
<td>ch5</td>
<td>synchronizing modules &quot;stiff&quot; and &quot;main&quot;</td>
</tr>
<tr>
<td>ch6</td>
<td>counting # of modules &quot;element&quot; have completed</td>
</tr>
<tr>
<td>ch7</td>
<td>synchronizing modules &quot;ns&quot; and &quot;input&quot;</td>
</tr>
<tr>
<td>ch8</td>
<td>synchronizing modules &quot;solver&quot; and &quot;main&quot;</td>
</tr>
</tbody>
</table>

#include "gl.c"

const A_num_iteration= 10;

main()
{
  int i,j,i_res;
  int shmid_GD, shmid_vom;
  GLOBAL_DATA *GD= new GLOBAL_DATA[1];
  VOM *vom= new VOM[1];
  int fdch1, fdch2, fdch3, fdch4, fdch5, fdch8;
  struct ID ID_to_stiff;

  //Non-blocking opening named pipes "ch1" and "ch2" for input module
  if((sizeof(GLOBAL_DATA)> A_pipe_size) || (sizeof(VOM)> A_pipe_size))
  {
    cout<<"Size of GD or vom is larger than pipe limitation."<<endl;
    cout<<"sizeof GD " <<sizeof(GLOBAL_DATA)<<" " <<A_pipe_size<<endl;
    cout<<"sizeof vom " <<sizeof(VOM)<<" " <<A_pipe_size<<endl;
  }
exit(1);

if((shmid_GD= shmget(GD_SHM_KEY, sizeof(GD), 0600|IPC_CREAT|IPC_EXCL)<0)
    perror("Shared memory ID for GD in main");
if((shmid_vom= shmget(VOM_SHM_KEY, sizeof(vom), 0600|IPC_CREAT|IPC_EXCL)<0)
    perror("Shared memory ID for vom in main");

// give info shmid_GD and shmid_vom to "input"
if((fdch1= open("ch1", O_RDWR|O_NDELAY))<0)
    perror("open named pipe ch1 in main");
if((fdch2= open("ch2", O_RDWR|O_NDELAY))<0)
    perror("open named pipe ch2 in main");

vom= (VOM*)shmat(shmid_vom, 0, 0);
GD= (GLOBAL_DATA *)shmat(shmid_GD, 0, 0);
GD[0].assign_Title("GLOBAL-LOCAL FINITE ELEMENT ANALYSIS OF COMPOSITES");

/*
cout<<"shmid_GD= "<<shmid_GD<<endl;
cout<<"shmid_vom= "<<shmid_vom<<endl;
cout<<"GD= "<<GD<<endl;
cout<<"vom= "<<vom<<endl;
*/

// give info ID_to_stiff to "stiff"
if((fdch4= open("ch4", O_RDWR|O_NDELAY))<0)
    perror("open named pipe ch4 in main");

cout<<"Is vom file on the disk? (1/0)(YES/NO)"<<endl; cin>> i;
if(i== YES)
    {vom[0].Retrieve();
cout<<vom[0];
    }

for(j= 0; j< A_num_iteration; j++)
    {
        if(j==0)
            {cout<<"This is the first iteration. "<<endl;

            // give info shmid_GD and shmid_vom to "input"
if(write(fdch1, (char*)&shmid_GD, sizeof(shmid_GD))<0) 
  perror("write ch1 in main");
if(write(fdch2, (char*)&shmid_vom, sizeof(shmid_vom))<0) 
  perror("write ch2 in main");

//to execute exec! and continue this program, a fork is 
//necessary since fork can make a copy of this program and 
//continue after execi is done. 7/29/92
if(fork()==0)
  execl("input", (char*)0);
else
{
  // using message passing to synchronize two processes 
  // this process will wait until the writing end of the 
  // pipe is opened in module "input" 8/12/92
  if((fdch3=open("ch3", O_RDWR))<0)
    perror("open named pipe ch3 in main");
  if(read(fdch3, (char*)&tl, sizeof(int))<0)
    perror("read ch3 in main");
  close(fdch3);

  cout<<"In Do you want to compute stiffness matrix?" 
     <<" (1/0)(YES/NO)"<<endl;
  cin>> i_res;
  if(i_res== YES)
  {
    ID_to_stiff.Num_ele_type= GD[0].get_Num_ele_type();
    ID_to_stiff.shmid_GD= shmid_GD;
    ID_to_stiff.shmid_vom= shmid_vom;
    if(write(fdch4, (char*)&ID_to_stiff, 
             sizeof(ID_to_stiff))<0)
      perror("write ch4 in main");
    if(fork()==0)
      execl("stiff", (char*)0);
    else
    {
      /* using message passing to synchronize two processes 
         this process will wait until the writing end of the 
         pipe is opened in module "stiffness" 8/13/92*/
      if((fdch5= open("ch5", O_RDWR))<0)
        perror("open named pipe ch5 in main");
      if(read(fdch5, (char*)&tl, sizeof(int))<0)
        perror("read ch5 in main");

      //cout<<"remove shared memeory for material 
       //properties"<<endl;
      shmctl(GD[0].get_Mat_shm_id(), IPC_RMID, 
             (struct shmid_ds*)0);
if(fork()==0)
    execl("solve", (char*)0);
else
    wait(0);
} // fork to "stiff"

/* using message passing to synchronize two processes
   this process will wait until the equation solver
   is done */
8/13/92 */

if(i_res==YES)
{
    if((fdch8= open("ch8", O_RDWR))<0)
        perror("open named pipe ch8 in main");
    if(read(fdch8, (char*)&t1, sizeof(int))<0)
        perror("read ch8 in main");
    close(fdch3);
}

cout<<"Do you want to run another iteration? (1/0)(YES/NO)"<<endl;

if(i==YES)
{
    // give info shmid_GD and shmid_vom to "input"
    if(write(fdch1, (char*)&shmid_GD, sizeof(shmid_GD))<0)
        perror("write ch1 in main");
    if(write(fdch2, (char*)&shmid_vom, sizeof(shmid_vom))<0)
        perror("write ch2 in main");
}

if(fork()==0)
    execl("input", (char*)0);
else
{
    // using message passing to synchronize two processes
    if((fdch3= open("ch3", O_RDWR))<0)
        perror("open named pipe ch3 in main");
    if(read(fdch3, (char*)&t1, sizeof(int))<0)
        perror("read ch3 in main");
    close(fdch3);

    cout<<"Do you want to compute stiffness matrix?"
    " (1/0)(YES/NO)"<<endl;
cin>>i_res;
if(i_res==YES)
{
    ID_to_stiff.Num_ele_type=GD[0].get_Num_ele_type();
    ID_to_stiff.shmid_GD=shmid_GD;
    ID_to_stiff.shmid_vom=shmid_vom;
    if(write(fdch4,(char*)&ID_to_stiff,
            sizeof(ID_to_stiff))<0)
        perror("write ch4 in main");
    if(fork()==0)
        execl("stiff",(char*)0);
    else
    {
        // see notes above
        if((fdch5=open("ch5",O_RDWR))<0)
            perror("open named pipe ch5 in main");
        j1= read(fdch5,(char*)&j1,sizeof(int))<0)
            perror("read ch5 in main");
        //cout<<"remove shared memeory for material
        //properties"<<endl;
        shmctl(GD[0].get_Mat_shm_id(),IPC_RMID,
               (struct shmid_ds*)0);
        if(fork()==0)
            execl("solve",(char*)0);
        else
            wait(0);
    }
}
else
    j=A_num_iteration;
}

cout<<"Do you want to store Version Manager Data? (1/0)(YES/NO)"<<endl;
cin>>i;
if(i==YES)
    vom[0].Store();

cout<<"Do you want to read [K]? (1/0)(YES/NO)"<<endl;
cin>>i;
if(i==YES)
{
    int Dim_total=GD[0].get_Ele_DOF();
    MATRIX K;
int size_K = sizeof(K)+Dim_total*Dim_total*8;

for(j=0; j<GD[0].get_Num_ele_type(); j++)
{
    cout<<"Type #"<<j+1<<endl;
    if(K.Retrieve("stiff.dat", j*size_K)== TRUE)
        cout<<"K[0][0], K[0][0]= "<<K[0][0]<<endl;
        cout<<"K[Dim_total-1][Dim_total-1]= "<<endl;
    else
        cout<<"[K] does not exist!"<<endl;
}

if(shmdt((char *)GD)<0)
    perror("detach shared memory of GD in main");
if(shmdt((char *)vom)<0)
    perror("detach shared memory of vom in main");

shmctl(shmid_GD, IPC_RMID, (struct shmid_ds*)0);
shmctl(shmid_vom, IPC_RMID, (struct shmid_ds*)0);

return 0;
APPENDIX E

DISSERTATION ABSTRACT

OBJECT-ORIENTED MODELS FOR
NUMERICAL AND FINITE ELEMENT ANALYSIS

By
George Gang Yu, Ph.D.
The Ohio State University, 1994
Professor Hojjat Adeli, Adviser

The major accomplishments of this Dissertation research are:

1. A new architecture has been proposed for numerical and finite element analysis. This architecture integrates blackboard architecture, enhanced entity-relationship model, finite element analysis, dynamic data management, version management, and concurrent programming in a uniform object-oriented programming paradigm.

2. An object-oriented enhanced entity-relationship model has been created for effective processing of a myriad of data types encountered in finite element analysis of complex engineering problems. An object-oriented FEA class library using database management techniques has been developed.
3. A new data management model for complex engineering data objects has been proposed. This model consists of a data storage structure and a 3+ index system. The integration of database management techniques with numerical analysis is particularly useful for managing large quantities of permanent, intermediate, and temporary data.

4. A version management model for numerical analysis has been proposed. This model is based on a proposed parallel version graph, which truly represents the characteristics of design objects with alternative data in a trial-and-error process.

5. With the basic synchronization techniques and interprocess communication capabilities, a concurrent computing environment is simulated on a UNIX based workstation. An efficient equation solver using object-oriented programming and iterative methods is developed under this environment.

6. The models and concepts developed in this research have been applied to solution of a complex engineering problem, interlaminar stress analysis of composite laminates.
APPENDIX F

INDEX

3+ Index System 104, 105
    illustration 106

ADC
    alternative data class 118

ADO
    alternative data objects 114
    attributes 88

BB
    a new model 143
    advantages 141
    blackboard architecture 51
    controller 140
    controlling data slots, 142
    knowledge source 140
    numerical data slots 142
    original framework 29
    world model 140

C++ 59
    call FORTRAN subroutine 85
    co-routine library 146
    fstream 94
    I/O library 59
    ifstream 59
    ofstream 59
    task library 146

CAE
    computer-aided engineering 53, 86
    extensive numerical analysis 113

class
    ELEMENT 74, 134
    ELEMENT_SEQ 73, 134
    GAUSS 70, 74
    GENERAL_DATA 65, 74, 134
    GLOBAL_ELEMENT 83, 134
    JACOB 69
MATERIAL 82, 134
MATRIX 63
NODE 70, 134
NODE_GL 81
Object_Record 128
OVM 130
SEMAPHORE 150
SHAPE 65
SHAPE_4 67
SHAPE_6 67
VECTOR 61
Version_Record 129
class card 60, 128
  component
    Class Name 60
    Composed Class 60
    Data 60
    Method 60
    Non-member Function 60
    Super Class 60
  components 60
  ELEMENT 76
  ELEMENT_SEQ 75
  GAUSS 70
  GENERAL_DATA 66
  GLOBAL_ELEMENT 83
  JACOB 69
  MATERIAL 82
  MATRIX 64
  NODE 71, 111
  NODE_GL 82
  Object_Record 129
  OVM 131
  SEMAPHORE 151
  SHAPE 67
  SHAPE_4 68
  SHAPE_6 68
  SPARSE_VECTOR 181
  VECTOR 62, 180
  Version_Record 130
class extensibility 56
class hierarchy
  ELEMENT 77
  GLOBAL_ELEMENT 84
code reusability 56
composite
  lamina 78
  laminate 78
  matrix 78
  computer-aided engineering, see CAE
COOFEA
Task Input 158
  task Node 161
  task Node_Seq 161
  Task Stiff 163

COOP
  Concurrent object-oriented programming 141,139
    Act 1 139
    Actor model 139
    ES-Kit 139

COOPL
  ConcurrentSmalltalk 139
  PC++ 139

data
  alternative 118
  atomic 114
  complex 89
  deletion 107
  dynamic component 99
  header 99
  insertion 106
  Length 99
  Offset 99
  redundancy 131
  retrieval 107
  storage structure for MATRIX 102
  storage structure for VECTOR 100
  un-fixed 118
  updating 107

data model
  EER 55
    entity-relationship model 88
  ER 53
    extended entity-relationship model 91
  object-oriented data management 92
  object-oriented data model 91
    approach
      behavioral approach 91
      operational approach 91
      structural approach 91

database
  central 92, 115
  manipulation language (DML) 53

DBMS
  column 88
  database management system 53
  instance 88
  relation schema 88
  tuple 88
  design class 114
  design object 114
Dissertation objectives 2

dynamic object array 97

entity-relationship (ER) data model, see ER of data model
extended entity-relationship (EER) model, see EER of data model

FEA
  element interpolation
    Q4 65
    T3 65
    T6 65
  finite element analysis 52
file manager 104, 105
file operation
  BOF 94
  close 94
  EOF 94
  open 94
  read 94
  seekg 94
  seekp 94
  write 94
  finite element analysis, see FEA

generalization 55
global-local finite element modeling 156
global-local modeling 55, 78

incremental node object 72
index card, see class card
inheritance 56
INTRODUCTION 1
IPC
  interprocess communication 149
  message queue 154
  pipe 150
  shared memory 155
iterative method
  Gauss-Seidel method 173
  Jacobi method 173
  Successive Over Relaxation method 174

monitor 139

object version manager, see OVM of version management
object version tree, see OVT
object-oriented database management system, see OODBMS
object-oriented finite element analysis, see OOFEA
OODBMS
  object-oriented database management system 113
OOFEA
class library 60, 158
object-oriented finite element analysis 60

OOP 1, 51
advantages for DBMS 56
class card, see class card

OVT
characteristics 120
derivative 116
object version tree 116
path 116
path to versions 117

parallel version graph, see PVG
PVG
illustration 124
parallel version graph 123

recursive dynamic object 97
relationship
aggregation 88, 90
inheritance 88, 90
kind-of 74
multiple-inheritance 74
part-of 53, 60

SIMULA-67 1
Smalltalk-80 1
specialization 55
speedup 165
synchronization graph 159
task E1 165
task Input 160
task Stiff 162

time stamp 118
trial-and-error 119

version management
a new moel
application 132
class identifier 125
object identifier 125
OVM
data structure 126
example 136
implementation 128
object version manager 125
Object_Index 125
Object_Record 125
Object_type 98, 131
operation 127
copy 127
I/O 128
retrieval 127
version creation 127
Version_Record 125
version identifier 125
alternative data 114
characteristics of versions in CAE 123
class (type) level 119
class inheritance 114
design object set 115
identifier 114
object (instance) level 119
time history 113
version
  inheritance 115
  revision 115
version definition 114
version management model 132