An object oriented design for finite element analysis

by Terrance E Kubat

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University
© Copyright by Terrance E Kubat (1992)

Abstract:
Finite element methods are used to solve a variety of problems in engineering. They have evolved into sophisticated numerical analysis techniques requiring powerful computer system solutions. Programs implementing these methods tend to be complex both in development and maintenance. With significant research and new developments in finite element technology, code evolution is a significant issue.

Finite element software is generally written by engineers using a procedural language like FORTRAN. This results in additional complexity to the system: the engineer's model of the problem must be transformed to meet the requirements of the programming language. Unfortunately the code obscures many of the concepts used in the finite element method.

It becomes difficult to see what the program is modelling, making system verification and modification that much harder. A programming system which captures the higher level concepts of the method and allows for this evolution is desired.

This thesis builds upon object oriented principles of design to create an extensible system for building finite element analysis programs. The design has been adapted from previous work done in the Common Lisp Object System. The C++ programming language is chosen as a more appropriate vehicle for implementing this engineering tool, which is a set of high-level finite element data types. Thede abstract data types define the objects in the engineer's model and therefore bring the software closer to the terminology used in finite element methods.

The result of this project is a base library for finite element analysis. This library provides a foundation upon which finite element applications can be built. An example program demonstrates the library for two dimensional structural trusses. This application is described and followed by a discussion of future library expansion.

N378
K9515

AN OBJECT ORIENTED DESIGN FOR FINITE ELEMENT ANALYSIS

by

Terrance E. Kubat

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

December 1992

APPROVAL

of a thesis submitted by

Terrance E. Kubat

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

11/25/92 — Ray S. Babcock

Date — Chairperson, Graduate Committee

Approved for the Major Department

11/25/92 — J. Denbigh Starkey

Date — Head, Major Department

Approved for the College of Graduate Studies

12/14/92 — R. L. Brown

Date — Graduate Dean

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission of extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signed

Date 25 NOV '92

TABLE OF CONTENTS

	Page
ABSTRACT	vii
1. INTRODUCTION	1
Problem	1
Goals	2
Scope	3
2. DESIGN OF THE FINITE ELEMENT LIBRARY	4
Introduction to the Finite Element Method	4
Numerical Analysis Technique	4
Mathematical Modelling	4
The Direct Stiffness Formulation	5
A Procedural Outline of the Process	7
Current Implementations	7
Object Oriented Design	8
Overview	8
Natural Classifications	8
The Process	9
An Object Oriented Look at FEA	9
Identifying Objects and Classes	9
Foundations	10
Base Classes	11
Truss Classes	14
Class Hierarchy	15
Types of Relationships	15
3. SOFTWARE DEVELOPMENT ADVANTAGES OF C++	17
Extensions from C	17
Operator Overloading	17
Constants	18
Reference Type	18
Default Data	18
Efficient Functions	19
Dynamic Memory	19
Object Oriented Features	20
The Class	20
Information Hiding	20
Inheritance	21
Modularity	21
Evolution	21
4. THE FEA LIBRARY INTERFACE	23
A Guide for Users of the Library	23
Class Description Format	23
Constructors and Destructors	23
Numbering	24
Alphabetical Class Descriptions	24

TABLE OF CONTENTS--Continued

Page

5. IMPLEMENTATION 29

Library Implementation 29
Beyond Design Objectives 29
Handling Errors 31
Incremental Construction Benefits 32
The Exemplar Program: Truss Analysis 33
A General Description and Outline 33
Specifying Input Data 33
How the Library is Used 36
An Example Problem Solved 38

6. CONCLUSIONS 40

Summary 40
Deficiencies in the Library 41
Possible Extensions 42

REFERENCES CITED 43

APPENDICES 46
A. C++ LIBRARY HEADER FILES 47
CoordSys.h 48
Dof.h 49
DSymMat.h 51
Element.h 52
FEM.H 53
global.h 54
Isotrop.h 55
Material.h 56
Node.h 57
Ortho.h 58
Point.h 59
TrusElem.h 60
TrusNode.h 61
B. INPUT AND OUTPUT FILES FOR EXAMPLE 62
Input Data 63
Output Data 64

LIST OF FIGURES

Figure — Page

1. Bridge Structure 5
2. Beam Model 5
3. Some element types 6
4. Coordinate Systems 10
5. Truss Element DOFs 12
6. Library class diagram 16
7. Outline of main() 34
8. Example input file. 35
9. Example truss model 39
10. Example truss results 39
11. CoordSys.h 48
12. Dof.h 49
13. DSymMat.h 51
14. Element.h 52
15. FEM.h 53
16. Global.h 54
17. Isotrop.h 55
18. Material.h 56
19. Node.h 57
20. Ortho.h 58
21. Point.h 59
22. TrusElem.h 60
23. TrusNode.h 61
24. Input Data 63
25. Output Data 64

ABSTRACT

Finite element methods are used to solve a variety of problems in engineering. They have evolved into sophisticated numerical analysis techniques requiring powerful computer system solutions. Programs implementing these methods tend to be complex both in development and maintenance. With significant research and new developments in finite element technology, code evolution is a significant issue.

Finite element software is generally written by engineers using a procedural language like FORTRAN. This results in additional complexity to the system: the engineer's model of the problem must be transformed to meet the requirements of the programming language. Unfortunately the code obscures many of the concepts used in the finite element method. It becomes difficult to see what the program is modelling, making system verification and modification that much harder. A programming system which captures the higher level concepts of the method and allows for this evolution is desired.

This thesis builds upon object oriented principles of design to create an extensible system for building finite element analysis programs. The design has been adapted from previous work done in the Common Lisp Object System. The C++ programming language is chosen as a more appropriate vehicle for implementing this engineering tool, which is a set of high-level finite element data types. These abstract data types define the objects in the engineer's model and therefore bring the software closer to the terminology used in finite element methods.

The result of this project is a base library for finite element analysis. This library provides a foundation upon which finite element applications can be built. An example program demonstrates the library for two dimensional structural trusses. This application is described and followed by a discussion of future library expansion.

CHAPTER 1

INTRODUCTION

Problem

As a numerical method, finite element analysis has been developing alongside the digital computers which enabled its practical usage. The finite element method has become a premier technique employed to solve a wide variety of engineering problems. Computer implementations of the finite element analysis (FEA) tend to be very large, complicated FORTRAN programs which obscure the mathematical and engineering concepts which define the method. This creates a problem for those who must maintain existing programs, as well as for developers wishing to expand capabilities to reflect recent advances in FEA technology.

The problem here is twofold. First, FEA ideas are obscured in a programming language which forces a transformation from the application domain into the digital machine's capabilities [Abelson 85]. FORTRAN was invented to solve this very problem, that is, to raise the level of abstraction in programming. In its time, and over the years it has proven to be a great success: allowing scientific formulas to be written out in a program which is more readable than the equivalent machine (or assembly) language program. It does not go far enough however, failing to capture higher level concepts required for today's complex applications.

Secondly, the organization of a typical FEA implementation is rigidly bound to the types of elements used and problems to be solved. Specifically, a change in the implementation of a data structure likely results in rewriting all modules which access that structure. As finite element analysis is still a very active area of research, this 'hard

coding' of modules and data types severely limits the extension of a program to provide advanced capabilities utilizing newly invented technologies. At the same time, developing a new implementation from scratch can be quite expensive in both time and money [Nagy 78].

Goals

The purpose of this thesis is to investigate a possible solution to these two problems. Ideally, a FEA computer program should read like a textbook on the method. Concepts, terminology, organization, and solution steps should all be preserved in the program's modules--at least on the surface, or in the interface. A programming language built upon the concepts of abstraction and inheritance is needed. Obviously, FORTRAN will have to be abandoned in favor of a language which will better support programming in the domain of the application. This creates somewhat of a dilemma however. Engineers are the developers, maintainers, and users of FEA programs and typically have only received training in procedural language programming: generally FORTRAN. While the best languages for raising the abstraction level of a program fall into the object oriented category, they generally require a very different paradigm.

Fortunately there are a number of hybrid object oriented languages available today which bridge this gap. The one chosen for this thesis is C++ [Cox 86] [Wiener 88]. Due to its very popular 'parent' language C, which has become somewhat of a universal procedural language, C++ has become a very popular language in recent years. Popularity aside however, C++ provides a solid base upon which to build a numerically efficient, yet readable, extendable library for finite element analysis.

Scope

It is intended here, that a prototype system of finite element data types be created and demonstrated. This system is meant to investigate, and demonstrate a base upon which a complete FEA implementation may later be developed. An exemplar program has been written to allow some experimentation with the ideas. No attempt has been made in addressing the issues of a user interface, pre- or post-processing. Similarly, flow of control issues resulting from parallelism or event-driven environments are not considered. The focus has been made on a representation of finite element concepts within the internal structure of the library.

CHAPTER 2

DESIGN OF THE FINITE ELEMENT LIBRARY

Introduction to the Finite Element Method

Numerical Analysis Technique

The finite element method is a computer oriented analysis technique used by engineers in a wide variety of application areas. It has been applied to problems ranging from heat conduction to fluid flow to structural analysis [Clough 89]. The method involves approximating the behavior of a continuous medium with an imaginary mesh of simple elements. Each element is defined by a small set of interconnection points called nodes. Nodes are defined to represent the boundary of the continuum and also represent arbitrary points within it. By choosing elements and their behaviors carefully, an approximation can be made to the behavior of the continuum. For the remainder of this chapter, a focus will be made on the displacement formulation of finite element analysis as used for a static, linear analysis of structures made of elastic materials.

Mathematical Modelling

In representing a mathematical model for structural analysis several items need to be addressed, including the geometry, the material properties, the boundary (or support) conditions, and the applied loads and forces. While in many cases 'exact' elasticity solutions have been formulated for various classes of structural problems, there still exist an infinity of problems for which no known solution exists. It is here where, finite element analysis allows practical numerical approximations to the solution.

To illustrate, consider the bridge shown in the figure 1. A truck, representing a load on the bridge is also shown.

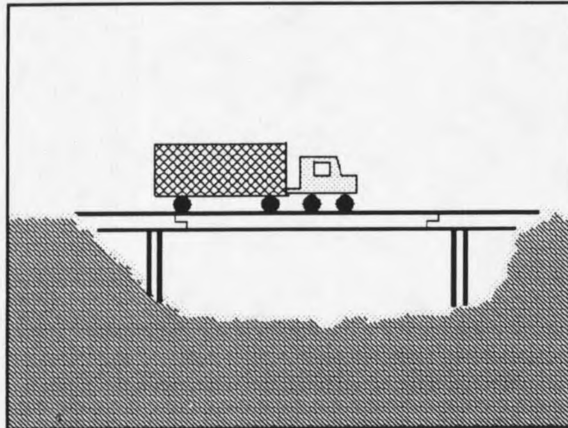

Figure 1: Bridge Structure

As the bridge is loaded, and deflects under the load it is desired to determine its behavior and the stresses it must withstand. (For this particular problem analytical as well as empirical results are readily available.) Figure 2 shows how this bridge might be modeled, with 5 beam-type elements and 6 nodes.

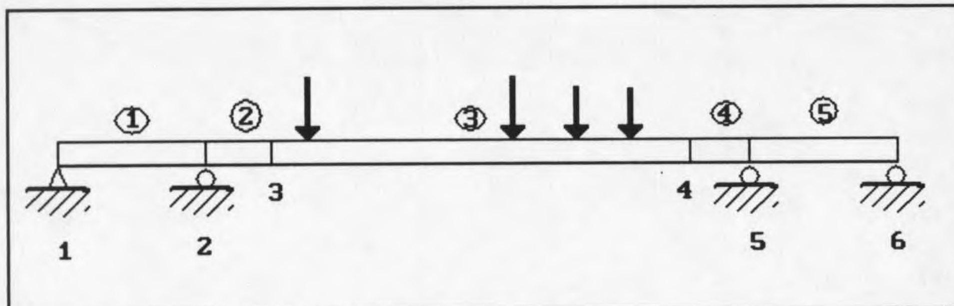

Figure 2: Beam Model

The typical analysis may be made more accurate by using either a larger number of small elements or higher order elements which can better simulate the overall behavior.

The Direct Stiffness Formulation

The method may be summarized in the solution of a system of equations of the following form: [K][d] = [F]. Where [K] is the

'stiffness' matrix of the model, [d] represents the nodal displacements (or degrees of freedom), and [F] is the set of nodal loads on the structure. The system is solved for the unknown, [d]. There are equations of equilibrium for each node in the model, and these are 'assembled' into one set of equations by summing member forces. Once the nodal displacements are found, the results are then fed back into each element to determine stresses within the element.

While the above characterization lends itself to a broad understanding of the method, it leaves out many significant details. As was mentioned earlier, FEA is used on a wide variety of problems; even within structural engineering there are many different classes of problems. The simple one dimensional beam element is just one of many types which have been formulated. Most are more complex, model two or three dimensions, have more than two nodes, or allow for curved boundaries. Some examples are shown in figure 3.

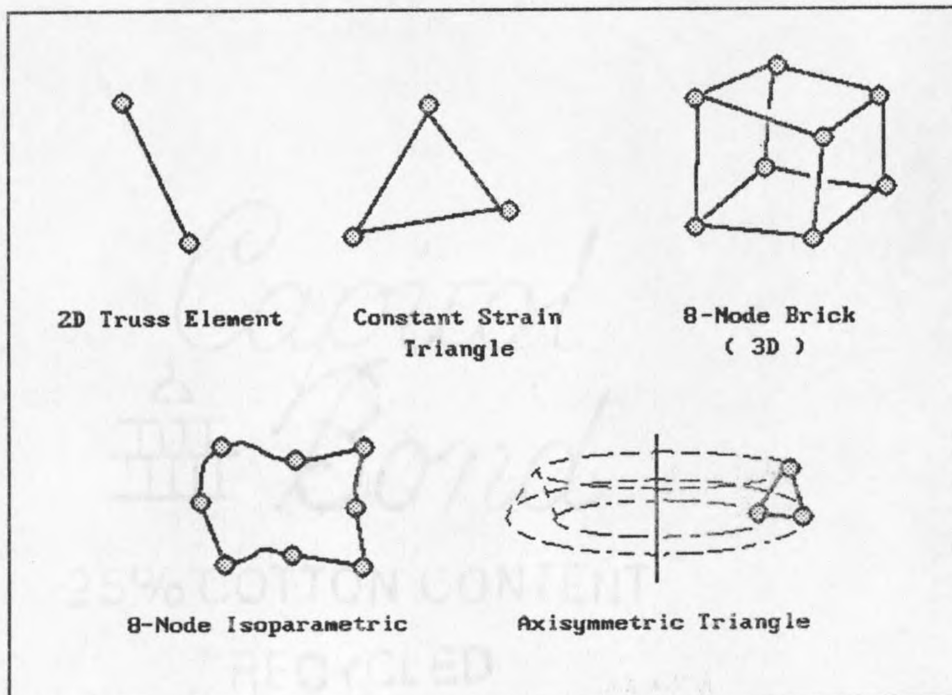

Figure 3: Some element types

Other complications are found in the geometry of the structure, the material used (i.e., steel or wood), and the types of loading. Elements are formulated according to their own 'local' coordinate system, which may in fact be curvilinear. Thus when, assembling the system of equations, or back-solving for element stresses, various coordinate transformations need to take place. Materials are represented mathematically via an elasticity matrix which is used in the calculation of an element's stiffness matrix--a process which requires numerical integration for some element types. Finally, loads which do not coincide with the nodes of the model must be converted into equivalent nodal loads before the system of equations may be solved. For a more rigorous treatment of the method, the reader is referred to the references [Weaver 84] [Zienkiewicz 89].

A Procedural Outline of the Process

General steps for static analysis. [Weaver 84]

1. Create Structure Model
    - A. Read Structural Data
        1. Problem Identification
        2. Material Information
        3. Nodal Information
        4. Element Information
    - B. Assemble Stiffness Matrix
        1. Determine Displacement Indices
        2. Generate the Elasticity Matrix
        3. For each Element
        4. Modify Stiffness for Restraints
2. Create Loading Case
    - A. Enter Loading Data
        1. Nodal Loads
        2. Optional: Other Load Types
    - B. Optional: Compute Equivalent Nodal Loads
        1. Directly or
        2. Using Numerical Integration
        3. Convert to Global Coordinates
    - C. Sum all Load Contributions at each node
3. Solve System of (Nodal Equilibrium) Equations
4. Calculate Results
    - A. Restrained Nodal Reactions
    - B. Element Stresses and Forces

Current Implementations

As mentioned above, current implementations of FEM programs are generally written in FORTRAN, and they are large, difficult to understand and thus hard to modify. Typically, when a new element type

is added to the program, there are a number of subroutines which need to be rewritten. These may be scattered throughout the program. In addition, the storage method for such items as symmetric matrices must be carried explicitly around inside the program. At worse, even the user must be aware of internal data structures as indicated by [Zienkiewicz 89], "The process of specifying the boundary conditions,...is tied to the method adopted to store the global arrays."

John Baugh and Daniel Rehak have done some work in addressing these concerns, and have implemented a system using CLOS (Common Lisp Object System) [Baugh 89A] [Baugh 89B] [Baugh 91]. However, this language, and its functional paradigm, has limited appeal to engineers firmly rooted in procedural based languages. Their ideas and the organization of their library have been used as a pattern for creating the design which follows.

Object Oriented Design

Overview

The object model of design is based on a number of important concepts including abstraction, encapsulation, modularity, and hierarchy [Booch 91] [Fenves 89]. Contrary to many cookbook style approaches to programming, object oriented design is an incremental process by which the application starts with a small and simple form and gradually evolves into the complex system that is required. The breakdown of a problem is facilitated by the natural classifications of objects and ideas in the domain of the application.

Natural Classifications

This is natural because this is how we deal with various layers of complexity in our own environment. For example, when looking at a park, the mind registers 'trees'. When in fact, trees come in all sorts of shapes, sizes and colors. They may be conifers, broad leaf, or even palm trees--a distinction which we choose to make only when necessary.

Each species has its own characteristics which may or may not be important in a given situation. These features are encapsulated within the abstract notion: tree. Similarly, there are often meta-levels in the hierarchy, as a tree is in fact just a plant (while not all plants are trees).

The Process

The process of object oriented design starts with identifying and classifying the domain according to an appropriate level of abstraction. Then, the semantics of the objects must be determined and defined, Finally these objects must be implemented [Booch 91]. This step results in a combined set of data structures and methods (or functions) which operate on the data. Each concept, idea, 'thing' or activity in the domain may be given its own class--a class which is clearly present in the computer program--with its own operations (which are hidden) and an interface (which defines what it is, and how it may be used). Again, this process is repeated as necessary. While uncovering these layers of complexity new features are discovered, perhaps a commonality is recognized, or additional classes are required. The next section shows one possible organization for an FEA library of classes.

An Object Oriented Look at FEA

Identifying Objects and Classes

The design of the finite element library, then, begins with an identification of objects in the domain. Obviously some type of element and node classes will be needed--as these are fundamental to the method. But a closer examination reveals that these are not the most primitive concepts. Indeed, elements and nodes can be quite complex and require a supportive host of classes. A few foundation classes will be needed in order to build the FEA class library.

Foundations

There are two general areas which need support beyond those data types which are built into most modern programming languages. The concepts of vectors and matrices are fundamental mathematical objects of FEA. These are very common tools and therefore much has been done by way of creating classes to capture vector manipulation and algebra. For this thesis project, an early decision was made to take advantage of an existing commercial library--rather than to re-invent similar classes. Rogue Wave's Math.h++ library was employed, with classes representing vectors, matrices, and linear algebraic routines.

The next foundation area is that of geometry. The model representing a true structure must be defined mathematically. Directions and locations of supports and loads, sizes and orientations of elements, as well as the positions of all nodes must be recorded. A simple two dimensional cartesian coordinate class was created for the exemplar system, as well as a coordinate system class which provided for transformations of points between different systems. See figure 4.

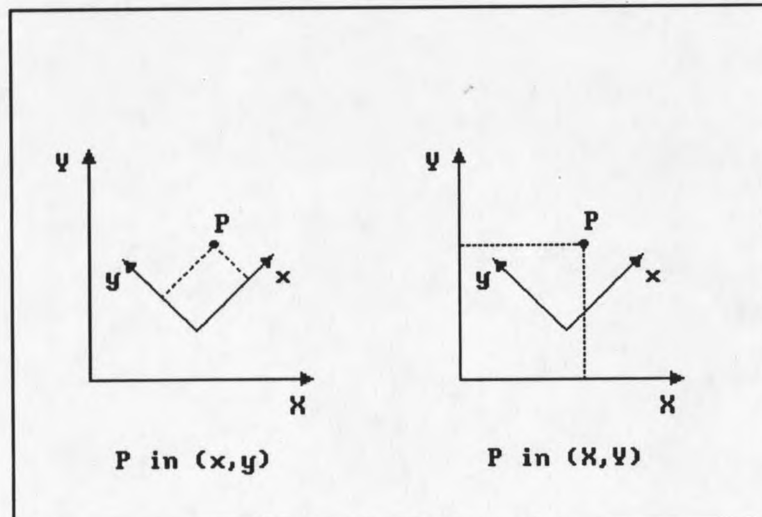

Figure 4: Coordinate Systems

Base Classes

With the foundation laid, an endeavor to capture the fundamental notions of the FEA method is made. This is accomplished with four classes. All of these are related, and the description and definition of each depends to some degree on the others. Perhaps the most independent, is the notion of a material model.

Materials. In the study of elasticity, a material's behavior is defined according to a set of constitutive equations. In the case of three dimensions, these equations relate the six independent stresses to the six corresponding strains. In two dimensions only three equations are required, along with a characterization of the problem type into plane strain or plane stress. This is the general form and can be shared by all specific types of materials--it is however too abstract. For any real material, more information is needed to describe fully these equations. This information comes from empirical values which are readily available for any given material. Furthermore, there are various classes of materials: isotropic, orthotropic, and anisotropic. Each of these, respectively, represents a more complicated behavior and requires more parameters to define. Each of these material types was assigned its own class, which inherits all the properties of the material base class yet exhibits its own specialized behavior.

Degrees of Freedom. Another basic concept embodied within the finite element method is the notion of a degree of freedom. A degree of freedom (DOF) represents the unknown variables at a node. A given node may have many unknowns. In the example given previously of the bridge model, each beam type element was shown with two nodes. What was not shown however, was that each node possessed three degrees of freedom. This is necessary in order to define the possible behavior of a simple beam. The figure below shows a truss element with its four degrees of

freedom, and a possible deflected shape reflecting deflections in two of these directions.

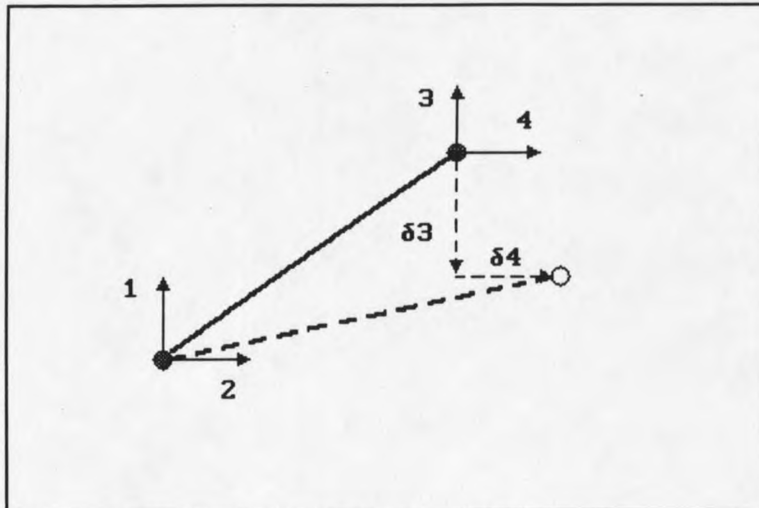

Figure 5: Truss Element DOFs

Degrees of freedom, may be free or supported. A structure model must be adequately supported to prevent its movement in space. A free DOF represents an unknown displacement value (in [D]) which is solved for in [K][D]=[F]. While a supported (or fixed) DOF means that the displacement is zero (or possibly another prescribed value) but the support reaction is unknown. The state of a DOF becomes very important to the correct assemblage of stiffness equations.

It is convenient to record loading information in the DOF as well; it represents an economy of space: because reactions or displacements are known [Zienkiewicz 89]. It is also where the load must be applied. So it is seen that a given node will possess one or more degrees of freedom which may be supported or have loads applied.

Nodes. A node represents a control point in the structural model. This point may be on a boundary, or in some arbitrary position within the structure itself. In most cases, a node lies on an edge or corner of an element and is used to ensure the compatibility of displacements between elements or between the model and a support boundary.

The node class is derived from a coordinate, as it represents a position in space, however it adds additional information and behaviors. A node is connected to one or more elements and for various reasons should store that information. The node also contains a number of degrees of freedom, but as this is variable, the node class itself remains abstract, no objects of type node are ever created. It will be shown how a practical node type may be derived from this base class.

Elements. An important class for capturing the concepts common to all element models is the abstract element class. This class defines the interface for a wide ranging category of element types. Elements may be one, two, or three dimensional. They may have one or more nodes which define their position, orientation, and shape. Each element has its own local coordinate system which makes it easier to calculate a stiffness matrix, compute equivalent nodal loads, and determine stresses from nodal displacements.

An element's stresses are a function of the displacement field within the element, which in turn is an interpolation of the nodal displacements. This approximate calculation is made by use of shape functions--usually a polynomial. The element local stiffness matrix is then calculated according to the formula:

$$B = \delta \cdot N \qquad k_e = \int_{V_e} B^T \cdot E \cdot B \, dV_e$$

N : Shape function matrix
B : Strain-displacement matrix
δ : Differential operator matrix
E : Elasticity matrix
k_e : element stiffness matrix
V_e : Volume of the element

For some of the simpler elements the stiffness matrix has been integrated exactly. However, for most elements the above equation must be approximated numerically.

Shape functions and integration are also required for elements which may have distributed loads, or other loads not coinciding with nodes. These loads must be converted to equivalent nodal loads before assembling the system of equations.

Truss Classes

The Exemplar Program. In order to demonstrate the object oriented design, a simple exemplar program has been written. This program supports the analysis of simple two dimensional trusses. (A truss is a structure made of pinned-end, axially loaded members which can only be loaded at the joints.) It shows how a system may be derived from the base classes to handle one specific element type.

TrussNode Class. A node for a two dimensional truss represents the meeting point for truss elements and possibly a structural support. Any number of elements may form a joint, and each joint has two degrees of freedom: horizontal and vertical displacement.

Deriving a truss node data type is quite straight forward, given the abstract node class. It is simply a matter of providing the appropriate algorithms for the already specified methods, and defining the node to have two degrees of freedom.

TrussElement Class. A truss element is an elastic bar which can be compressed or stretched. The strain, stress and force in the bar are functions of the distance a bar is stretched or compressed. This element needs two truss type nodes, one at each end. The material used for a truss element is isotropic, and each member holds a length and cross sectional area.

Once again, deriving the TrussElement from class Element is quite simple, its implementation made using algorithms which have been described in the references [Zienkiewicz 89].

Class Hierarchy

Types of Relationships

There are two types of relationships used in the FEA library of classes. These are referred to as ISA, and HASA [Booch 91]. The ISA relationships is a categorization relationship. In the example given earlier: an Oak ISA broad leaf tree ISA tree ISA plant. Thus ISA relationships represent an abstraction hierarchy.

The other type of relationship is the HASA. This represents a containment or a using arrangement. For example, a node contains a number of degrees of freedom, but is derived from a coordinate. Thus a node ISA coordinate type, while a coordinate is not a node. And each node HASA set of DOFs which are used to keep track of loading and displacement information.

The following diagram introduces the class names used in the library and clearly shows the ISA relationships between each class. Complete descriptions of the class interfaces are found in chapter 4 along with the header files in appendix A.

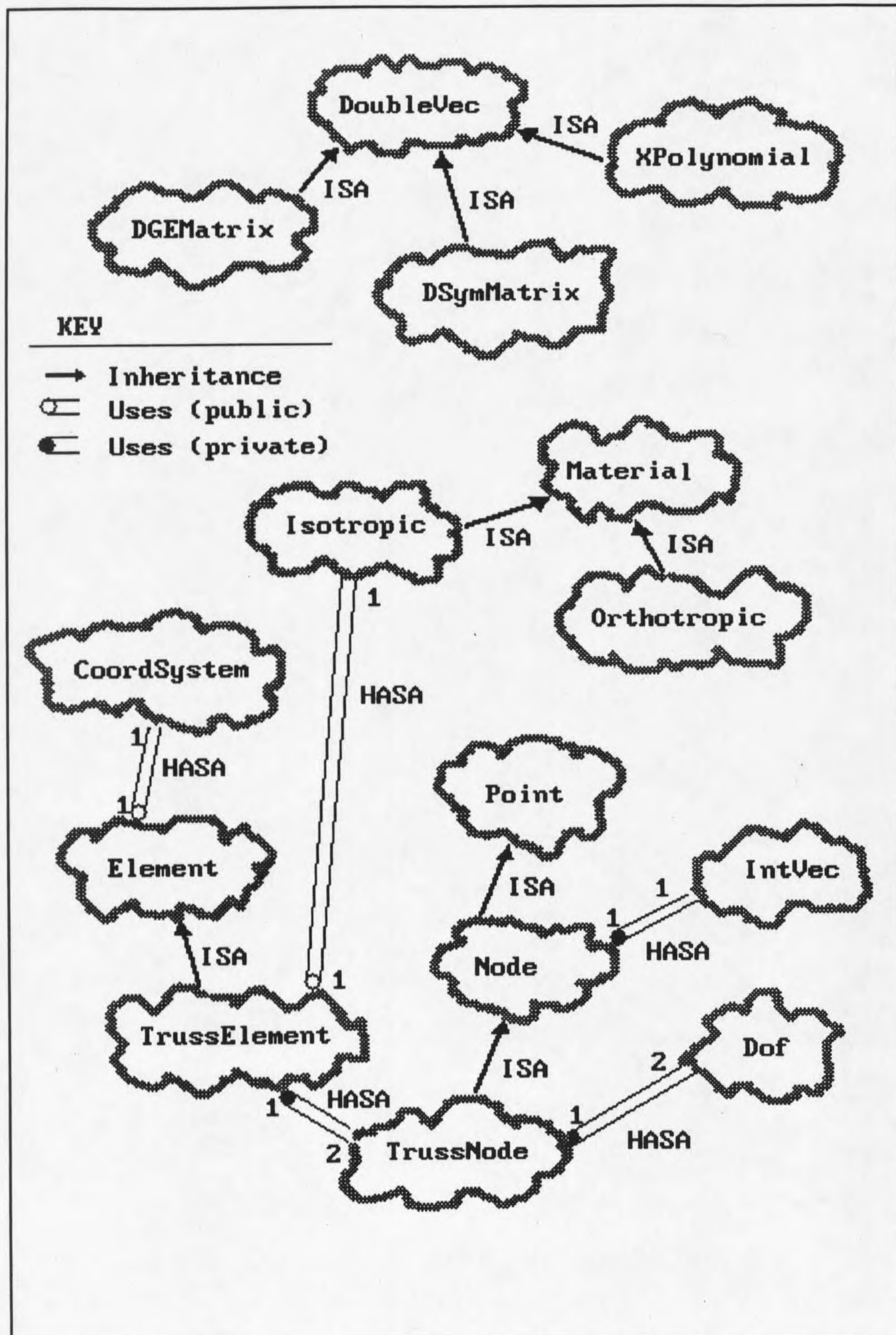

Figure 6: Library class diagram

CHAPTER 3

SOFTWARE DEVELOPMENT ADVANTAGES OF C++

C++ provides a number of features which make it especially useful for the implementation of this project. Like its predecessor, C++ is very efficient. With very little run-time overhead, except as requested by the programmer, it is well suited to numerical applications. In addition to this, C++ retains many of the same procedural language constructs familiar to engineers. While there are many solid references for C++ and its use, some of the most important concepts will be discussed here in relation to how they are employed in the library [Stroustrup 91] [Jordon 90] [Horstmann 91].

Extensions from C

Some of the extensions C++ provides do not directly relate to its object orientation. These features are provided to improve code readability, allow better type checking, make function calls easier, or enhance performance.

Operator Overloading

Function names can be overloaded in C++. This allows the reuse of common names for common operations. In addition to this, C++ allows most of the built in operators to be overloaded as well. By overloading the standard operators to manipulate higher level types, a simpler syntax can result. This tends to work well for mathematical types such as vectors and matrices. But there are also advantages for other types of operations. In the implementation of finite element library, the output operator '<<' is overloaded to allow transparent use by all the

classes. Each class has its own function to print the object on a stream, and thus any object may be printed in a standard C++ way.

Constants

C++ adds constants with the 'const' keyword. Used in declaring formal parameters to functions, const makes explicit which parameters are to be modified and which are guaranteed to be unchanged. In addition to this use, entire member functions within classes may be declared as constant. This means that they do not modify the object for which they are called. Constants are generously employed in the definition of classes to enhance their self documenting nature. (See Appendix A.)

Reference Type

The reference type was introduced into C++ to provide a syntactically cleaner use of pointers. References become an alias for the object they point to. References may be used as an 'lvalue' (on the left side of an assignment statement). Because they can be returned by functions, a function may be used as an lvalue. This is important for some of the overloaded operator functions. References are also very powerful when combined with 'const' for function parameters. By allowing efficient use of large data structures and the safety of pass by value parameters, constant references are also used throughout the library.

Default Data

C++ functions may also be defined to include default data for parameters. This feature seems trivial and yet has proven desirable within the library. For example, consider the material models used in finite element analysis. For isotropic materials, which are uniform in all directions, Young's modulus of elasticity is a constant used in the stress-strain relationships. For orthotropic materials, which have a layered makeup, two constants are necessary. In the implementation of a

material class it is convenient to have a single function 'youngsModulus' which returns one of these constant values.

For the isotropic class, no parameter is necessary, as a default parameter is supplied to the function. Similarly, for the orthotropic class, the same function may be used, with a direction parameter supplied by the user.

Efficient Functions

'Inline' functions are used in place of '#define' macros. The function body is placed in the code whenever the function is called. This technique is used to eliminate the function call overhead, and results in faster program execution. The potential drawback is an increase in size for the compiled code due to the expansion at each call location. Inline functions are used in the library for the short constant selector type functions whose purpose is to return a data value held by the object. Without inline functions, the C++ library would be very inefficient.

Dynamic Memory

One further enhancement C++ adds is a simpler mechanism for handling dynamic memory. The keywords 'new' and 'delete' are used in place of the traditional calls to malloc and free. This method provides a cleaner syntax and type safety (an explicit cast from a pointer is not required). The new operator can also be supported by a user defined error handling function. This mechanism proved useful in classes requiring data structures of variable size. For example, in the material class it was desirable to hold a descriptive name along with each object created. Because the name is provided at run time, 'new' is used to allocate space for the character array.

Object Oriented Features

The previous discussion revolves around the relatively minor improvements and extensions C++ makes to C. The most important contributions C++ makes to this project rest in the object oriented features. These are the features which will allow the library to harness the key elements of object oriented design: abstraction, inheritance, modularity, and evolution.

The Class

The key construct in C++ is the 'class'. Classes allow the definition of new types within the language. These user-defined types are not simply data structures, but encapsulate an object's identity and behavior, effectively simulating the semantics for built-in types. The class provides a means for instantiating objects of the type, ensuring proper initialization and destruction, and controlling access to critical data. The finite element library is built as a set of interrelated classes.

Information Hiding

The class is the mechanism for abstraction, holding inside the information necessary for its implementation. Each class is composed of two parts: the interface and the implementation. The interface functions as a declaration, which the compiler uses to allocate storage space and provide type checking. It also declares what information is hidden and what is visible. In fact there are three levels of protection: public, protected, and private. Users of the class need only see the public information--this is the interface. Private data is for the implementation only. And protected information is hidden from users but accessible to derived classes. By separating the various levels of concern, classes accommodate our natural powers of abstraction. The important information is made public, while the

details are kept out of sight unless needed. This makes the code easier to understand and maintain.

Inheritance

Classes may represent a single entity or idea that stands alone. However, the real power of classes in C++ is harnessed through derivation. A class may inherit from an existing class. This means that one class can be a specialization or generalization of another.

Consider once again the material models used in finite element analysis. Each model is represented by an elasticity matrix, [E], which is defined according to the physical constants of the model. This generalization (and others) can be captured into a class 'Material'. Now having this notion encapsulated, the specialized material models for isotropic, orthotropic, and anisotropic materials can be built from this base. The technique allows reuse of code that has already been written, while specifying a method for overriding parts that need specialization. In C++ this is done with 'virtual' functions.

Modularity

C++ programs demonstrate their modularity, by separating interface from implementation, by encapsulating concepts into classes, and by separating the layers of abstraction. With this organization, a change made in the implementation of a class is isolated from any code which uses the class--only the class needs recompilation. Similarly, new classes can build on the old ones (allowing code reuse) without modifying them and can be linked into existing programs without changing them. Only changes in the interface of a class result in widespread change, and these can be minimized with proper design.

Evolution

It is the combination of these object oriented features, along with the extensions listed previously that enable this flexibility, and ease of maintenance. The finite element method is a developing area,

experiencing significant change due to current research. The use of C++ in representing the basic concepts of finite element analysis, should help ease the maintenance problems which result from this evolution.

CHAPTER 4

THE FEA LIBRARY INTERFACE

A Guide for Users of the Library

Class Description Format

The following pages provide descriptions of each of the classes in the FEA library. Each class has a general description of its purpose and organization along with descriptions of all public member functions. The member functions are divided into two types: selectors, and modifiers. Constructors, which allow for the creation of a new object are not listed here, but are shown in the header file for the class. (See Appendix A.) Selectors are constant functions employed to get information from an object. The object is never modified by a selector function. Manipulator functions change the state of the object in some way.

These descriptions, along with the header files which provide function call parameters and return value syntax, represent the interface to the library.

Constructors and Destructors

A general note regarding the construction and destruction of an object is in order. In general, a declared object is an initialized object. The constructors provided require parameters such that an object can be fully operational as soon as it is declared. An exception to this rule applies to some of the classes when an array of objects is declared. In this case, the default constructor is called on each object of the array. The user should be careful to then call the

appropriate initialization function(s) before attempting to use the objects in the array.

Similarly, destructors for objects rarely need to be called directly. Global and automatic variables, as well as function parameters and temporary objects will all have their destructors called implicitly--it is arranged by the C++ compiler. However, objects declared dynamically using 'new' become the programmer's responsibility and should be destroyed using 'delete' when no longer needed. This will ensure proper behavior of the library as well as keep memory usage to a minimum.

Numbering

The FEA method has been described as a bookkeeping system. Elements, nodes, and degrees of freedom are all numbered. This numbering system is important insofar as it coordinates the various equations in the system. Classes which take an identification number in their constructors have been designed such that the user may use the counting numbers (1, 2, 3,...) when declaring these objects. This warning is given to avoid potential problems from the standard C way of loop control using the natural numbers (0, 1, 2,...). Objects created with a default constructor are given the identification number of -1, for easy detection.

Alphabetical Class Descriptions

CoordSystem. A coordinate system represents a reference point and an orientation relative to a global system. By definition, this global system is at the coordinate (0,0) and has an orientation of 0°.

CoordSystem	
SELECTORS	
angle	Returns the orientation angle.
transpose	Returns the transpose of the orientation matrix.
orientation	Returns the orientation matrix.
origin	Returns the reference point.
mapToGlobal	Returns a point's coordinates in global system.
mapToLocal	Returns a point's coordinates in this system.
printOn	Displays the coordinate system on a stream.
MODIFIERS	
operator =	Allows assignment from another CoordSystem.

DGEMatrix. The Double General Matrix represents a two dimensional array of doubles. This class is part of Rogue Wave's Math.h++ library and is fully documented in their manuals [Rogue Wave 91]. It allows the use of a matrix as if it were a built in class, similar to int, char, or double.

Dof. A degree of freedom (DOF) is a numbered unknown in the structure. It holds state information on whether the DOF is free, fixed or prescribed. It is also used to store any load on the DOF, as well as the resultant displacement. The DOF class is used by elements; the user need not be aware that it exists.

Dof	
SELECTORS	
idNumber	Returns the internal identification number.
isFixed	If dof is fixed returns true.
isFree	If dof is free returns true.
isPrescribed	If dof is prescribed returns true.
displacementValue	If dof is not free, returns displacement.
loadValue	If dof is free, returns load value.
assemble	Places dof data into system of equations.
printOn	Displays the dof on a stream.
MODIFIERS	
operator =	Assignment.
addLoad	Adds a load to any current loads.
setDisplacement	Sets the displacement value.
reset	Clears load or displacement values.

DoubleVec. The Double Vector class represents a one dimensional array of doubles. This class is part of Rogue Wave's Math.h++ library and is fully documented in their manuals [Rogue Wave 91]. It allows the use of a vector as if it were a built in type, similar to int, char, or double.

DSymMatrix. The Double Symmetric Matrix provides a more efficient implementation of a matrix that is known to be symmetric. It may be used anywhere where a DGEMatrix is expected.

DSymMatrix	
SELECTORS	
operator ()	Returns the matrix value indexed.
operator []	Returns the vector indexed.
row	Returns the indexed row of the matrix.
col	Returns the indexed column of the matrix.
product	Returns the cross product with the argument.
rows	Returns the size of the matrix.
cols	Returns the size of the matrix.
copy	Returns a copy of the matrix.
printOn	Displays the matrix on a stream.
MODIFIERS	
resize	Adjusts the matrix to a new size.
operator =	Assignment (sizes must be the same).

Element. An element is the abstract base class which defines the operations required for derived element types.

Element	
SELECTORS	
idNumber	Returns the internal identification number.
numNodes	Returns the number of nodes per element.
localCoord	Returns the local coordinate system.
printOn	Displays the element on a stream.
findStresses	Returns element stresses.
findStrains	Returns element strains.
localStiffness	Returns the local stiffness matrix.
MODIFIERS	
assembStiffness	Assembles the local stiffness into the global stiffness matrix provided.
operator =	Assignment from another element.

Isotropic. An Isotropic material is uniform in all directions. It can be specified with just two constants: Young's modulus of

elasticity and Poisson's ratio. This class inherits functions from class Material.

Material. Material is the abstract base class representing the common features of all material models. A model may be scalar, planar, axisymmetric, or solid. A planar model must be specified as either plane stress or plane strain. The member functions represent the common interface for all material classes.

Material	
SELECTORS	
name	Returns the string name of the material.
idNumber	Returns the internal id number.
eSize	Returns the size of the elasticity matrix.
printOn	Displays the material on a stream.
youngsModulus	Returns a Young's modulus value.
poissonsRatio	Returns a Poisson's ratio value.
e	Returns the elasticity matrix.
c	Returns the inverse of the elasticity matrix.
MODIFIERS	
operator =	Assignment.

Node. A node represents a coordinate point with degree of freedom and element connectivity information. This is an abstract class.

Node	
SELECTORS	
idNumber	Returns the internal id number.
numElements	Returns the number of elements connected.
numDofs	Returns the number of DOFs per node.
idElement	Returns an element's id number.
assemble	Assembles Node into the system of equations.
load	Returns the value of a load on a given DOF.
displacement	Returns the value of a displacement of a DOF.
printOn	Displays the node on a stream.
MODIFIERS	
connectElement	Connects the given element to this node.
operator =	Assignment.
disassemble	Records results from the global vector [D].
makeSupport	Fixes the appropriate degree of freedom.
applyLoad	Adds the load value to the correct DOF.
applyDisplacement	Sets a prescribed displacement to a DOF.
clearLoads	Resets the DOFs for a new loading case.

<u>Orthotropic.</u> An orthotropic material is one that is stratified. Up to eight constants are required to define this type of material. The principle and secondary directions are defined as 1 and 2, respectively. This class inherits functions from class Material.

<u>Point.</u> A point represents a position in space, and is specified with cartesian coordinates.

Point	
SELECTORS	
x	Returns the cartesian x coordinate.
y	Returns the cartesian y coordinate.
distanceTo	Returns the distance to the specified point.
printOn	Displays the point on a stream.
midPoint	Returns the midpoint of two points.
operator +	Returns the addition of two points.
operator -	Returns the subtraction two points.
MODIFIERS	
operator =	Assignment.
readFrom	Reads formatted coordinate values from a stream.

<u>TrussElement.</u> The truss element has two nodes, four degrees of freedom. It behaves much like a spring. A truss element is specified with an id number, two TrussNodes, a reference to its material type, and a cross sectional area. It supports all of the functions of an Element in addition to the following.

TrussElement	
SELECTORS	
area	Returns the cross sectional area.
length	Returns the length of the element.
material	Returns the material reference.
axialForce	Returns the axial force in the bar.
axialStress	Returns the axial stress in the bar.
axialStrain	Returns the axial strain in the bar.

<u>TrussNode.</u> A truss node implements the abstract Node, with two degrees of freedom. It provides all the functions specified in class Node.

CHAPTER 5

IMPLEMENTATION

Library Implementation

Beyond Design Objectives

The transition from design stage into an implementation in C++ requires some discussion. There are a number of areas where the language of implementation molds decisions, and effects readability, extensibility, and performance. While the design specifies minimum objectives and goals it does not specify exactly how those goals are to be achieved within a given programming language. The implementation must go beyond the basic functionality of the objects and classes and deal with the organization of source files, the creation and destruction of these objects and other peripheral design concepts.

It was stated previously that FEA programs typically entail extensive user interfaces, preprocessors and postprocessors. While the library itself makes no assumptions about these pieces in the overall scheme, it must be created in such a way that will not preclude them. To this end, all classes were designed without extensive input or output capabilities. Simple constructor functions create each object from raw data or previously created objects. Selector functions exist for querying an object's internal state. A 'printOn' method is provided which will present the object in user readable form on a given stream. Thus a very simple application may be created, sending output to the standard output device, or a postprocessor may query objects using the selector functions and then format its own presentation of the information.

<u>File Organization.</u> Source files in C++ follow the C convention: header files contain declarations while source files contain complete definitions. In the case of the FEA library of classes, each class consists of one header file and one source file. The header file is the class declaration containing prototypes for all member functions. It is used as an interface and reference for class users. In addition, all dependencies are declared, private members and functions are declared, and related function prototypes are given. While the header file is given a '.h' extension its related source file has the same name but with a '.cpp' extension. For example the class ELEMENT is declared in ELEMENT.H, and defined in ELEMENT.CPP. For many classes the file name is abbreviated (to eight characters or less) due to the operating system limitations.

<u>The Canonical Form.</u> In order to provide a uniform and sufficiently flexible implementation each class is constructed upon a standard canonical form [Coplien 92]. This format provides for creation, assignment and destruction of objects by requiring the implementation of four special member functions. These functions include two constructors, an overloaded assignment operator and a destructor. For example, the class POINT is provided with the following four functions:

1. Point::Point()
2. Point::Point(const Point &p)
3. Point::operator = (const Point &p)
4. Point::~Point()

The first of these is a default constructor which allows for an object to be declared without any initializing parameters. For most classes in the library this will not be used except in the case of declaring arrays of objects where the language prohibits the use of a more specific constructor. The second constructor is termed a copy constructor and allows the creation of one object from a previously created one. A copy

constructor handles two situations: passing objects to functions by value, and creating temporary objects required by the compiler. The overloaded assignment operator (number 3 above) allows redefinition of an object with a simple assignment statement. And finally a destructor is declared to handle the release of any dynamically allocated memory within the class.

Inline Functions. Other items of practical interest are contained within the header files. The first of these are the inline function definitions. Inline functions are provided for short functions (usually the selectors) for better readability as well as for performance enhancement. Where the function simply returns the value of a data member it is defined on the same line as the declaration. In a few cases, to provide a cleaner appearance, these functions are defined in a separate section near the bottom of the header file. In any case, inline functions are to be ignored in the interface. The only reason they are listed here is for the compilation process.

Related Functions. One final item found in the header file is a section of related functions. These are functions which operate on the associated class, but are not member functions. Every class in the library, for example, has an overloaded output operator '<<' which allows for transparent use of the printOn function. Any object created from the library may be used in a statement of the following form:

```
cout << theObject;
```

This statement will result in an inline call to:

```
theObject.printOn( cout );
```

while being easier to read.

Handling Errors

Another implementation decision was made with respect to error handling. It was decided that for the prototype system the simple method of sending an error message to 'cerr' followed by program

termination was the best method. A better alternative may be to call a user supplied error function, but for the simple system under consideration this approach was overcomplicated.

With proper construction ensured by class constructors, very few problems are encountered with bad or missing data. Those error conditions that do exist are centered around the DOF and Node classes. For example, the program will abort an attempt to return the displacement of a node for which the disassemble member function has not been called. In situations where it is possible, a warning may be issued and the requested action is ignored.

Incremental Construction Benefits

Once the design was completed, the implementation of the library proceeded rapidly. Starting with the foundation classes, each class was written and then tested separately. As each class was finished the three files (header, body, and test driver) were moved out of the development directory into protected areas.

This scheme had the advantage that once a type had been created and tested, the implementation may be forgotten--only its interface needs to be accessible. This allowed for a very clean development cycle: The body of a class seldom needed rework after the first implementation was tested.

There were a number of occasions, however, where the interface of a class was found to be inadequate after it was implemented. This was typically the case for the abstract base classes. In particular, the Material class needed to be rewritten when implementing the Element class. It was desired for the Element to hold a reference to a Material, but when implementing the TrussElement (which actually uses Isotropic materials) it was found that not enough functionality was built into the Material base class to access the Isotropic information.

This was remedied with a more complete set of pure virtual functions, which allows a more flexible use of derived classes.

The Exemplar Program: Truss Analysis

A General Description and Outline

Figure 7 is an outline of the exemplar program. Using C++ syntax it shows clearly how easy it was to implement the truss program using the FEA library. The most difficult part of the implementation was centered in the input processing functions, as these were required to parse a loosely formatted text file of input data. Ellipses in the outline indicate details which have been omitted for clarity.

Specifying Input Data

The exemplar program takes as its input a simple text file description of the problem to be solved. This file must be in the following format, typical of current FEA programs. The words appearing in bold are keywords recognized by the input processing functions and must appear as shown. The file is organized into three ordered sections: 1. header, 2. structure description, and 3. loading case descriptions. Comments are allowed at the beginning of each section and must have an asterisk as the first character on the line. Comments may also appear on the same line following keywords, as shown in the example data file of figure 8 [Laible 85]. Each of the three sections is described below.

Header. The header contains an overall enumeration of the problem type and size.

Structure Description. The structural description must indicate the positions of each node, and which of the nodes are supported. It also must describe the materials used. Each element is then identified by its corresponding nodes and material along with any additional information required. In the case of truss elements a cross sectional area is the only additional data.

```
#include <fem.h>
...
int numDimensions, numElements, numNodes, numSupports,
    numMaterials, numLoadCases;
char problemName[LINE];

int main( int argc, char *argv[] ) {
   ...
   ifstream input( argv[1], ios::in );
   ...
   TrussElement  *bar[MAXELEMENTS];
   TrussNode     *joint[MAXJOINTS];
   Isotropic     *material[MAXMATERIALS];

   readHeader( input );
   readTruss( input, bar, joint, material );
   echoTruss( cout, bar, joint, material );

   DGEMatrix K( 2*numNodes, 2*numNodes, 0.0 );
   int i;

   for( i = 0; i < numElements; i++ )
      bar[i]->assembleStiffness( K );

   for( i = 0; i < numLoadCases; i++ ) {
      DGEMatrix K2 = K.copy();
      DoubleVec F( 2*numNodes, 0.0 );
      DoubleVec D( 2*numNodes, 0.0 );
      int j, lcase;

      for( j = 0; j < numNodes; j++ )
         joint[j]->clearLoads();

      lcase = readLoadCase( input, joint );

      for( j = 0; j < numNodes; j++ )
         joint[j]->assemble( K2, F );

      D = solve( K2, F );

      for( j = 0; j < numNodes; j++ )
         joint[j]->disassemble( D );

      printResults( cout, bar, joint, D, lcase );

   }  // end LoadCase loop

}  // end main()
```

Figure 7: Outline of main()

```
* Example Truss Input Data File
*
* HEADER SECTION
*
Structure: Plane Truss Example 1.1
Dimensions 2
Elements   5
Nodes      6
Supports   3
Materials  1
Load Cases 2
*
* TRUSS DESCRIPTION SECTION
*
Coordinates  (user ID, X, Y)
1    0.0  0.0
2   10.0  7.5
     ...
*
Supports (node ID, direction)
1    X
1    Y
6    X
*
Materials (user ID, user name, Modulus of Elasticity)
1    Steel  29.0e6
*
Elements (user ID, node #1, node #2, material ID, area)
1    1   2    1    0.25
2    2   4    1    0.37
     ...
*
* LOADING DESCRIPTION SECTION
*
Load case 1  (node ID, action, direction, magnitude)
4   Force  Y  -3.2
6   Displacement  X  0.01
End load case 1
```

Figure 8: Example input file.

Loading Cases. Loading cases for truss problems are quite simple. Each loading case consists of a list of forces or prescribed displacements. Each of these occurs at a given node with a specified direction. Each loading case must terminate with an 'end' statement. An additional 'end' may be placed at the end of the file, but is not required.

How the Library is Used

ReadTruss Explained. The first significant use of the library is made in the function readTruss. Having obtained the general size of the problem to be solved, readTruss starts gathering specific structural information--starting with coordinate data. As each line is read from the file in this section the following line is executed:

```
n[j-1] = new TrussNode( j, x, y );
```

This statement is dynamically allocating a TrussNode object with the newly parsed input data. The TrussNode constructor is being called to initialize itself with the given parameters which reflect the user's identification number, and the cartesian coordinate locations of the node. A TrussNode is initially defined to be a free node, as most nodes in a structure will not be supports. Finally the address of this node is assigned to the array of node pointers. Once the coordinates have been read and all the nodes created support data is processed. Structural supports become fixed degrees of freedom in the numerical model and this information is handled by the TrussNode class. The following statements read the user data and call the appropriate TrussNode method for this action:

```
scanf( buffer, "%u %c", &j, &dir );
if( dir == 'X' ) n[j-1]->makeSupport( Node::x );
...
```

Here it is seen that makeSupport is called for the object pointed to by n[j-1] in the array of pointers to TrussNodes. The parameter to the function is defined in the interface to the Node class as an enumerated type--hence the scope qualifier Node::. This arrangement makes it clear which direction will be supported. In this case a simplified syntax would result from a makeSupport method taking a character argument, however the approach taken was chosen for uniformity in the library. In addition, the use of an explicitly labeled type makes it very clear what the parameter is intended to do and hence provides a greater readability for TrussNodes and any other types derived from the Node class.

The next section creates the material models from the input file data. This is accomplished in a manner similar to that of creating nodes. The following statement allocates the objects:

```
m[j-1] = new Isotropic( j, mName, Material::scalar, x );
```

The use of a qualified enumerated type is supplied as a parameter to the constructor, to indicate the appropriate one dimensional material model.

Elements are now created using the input data and the existing nodes and materials. The user supplies the element connectivity information, material identification number and an area for each element. The Element hierarchy is set up to use references (or pointers) to Nodes and Materials rather than to contain those objects. This allows for greater modularity, while still providing access to member functions for those objects. The constructor is called dynamically:

```
t[j-1] = new TrussElement( j, *m[mn], *n[nd1], *n[nd2], a );
```

<u>The Loading Case Loop.</u> Once the truss model has been created, solutions to various loading cases may be made. In the function main it is seen that the global stiffness matrix is declared and initialized to zeros. Each of the elements is then 'assembled' to create this matrix representing the entire structure.

As the loop is entered, a copy of this stiffness matrix is made (it is modified by the assembly of nodes) and then the vectors F and D are created anew for each loading case. In a final preparation to solving the next situation, all previous nodal loads are forgotten using the method clearLoads.

The readLoad function converts the input data into nodal loads and displacements in the model. This information is 'assembled' into the stiffness matrix [K2] and load vector [F] to complete the system of equations which is subsequently solved.

The result is the displacement vector [D], describing the action of each degree of freedom in the structure. This information is then

passed back to each of the TrussNodes using the disassemble member function.

Generating the Output. The function printResults sends the results to the standard output. This very simple function echoes the joint information (now with applied loads and resultant displacements) using the overloaded '<<' operator:

```
out << *n[i] << endl;
```

Following this object oriented display of results a more traditional table of nodal displacements is made. This requires a bit of formatting and the use of multiple function calls.

```
out << (n[i]->idNumber()+1) << '\t';
out << n[i]->displacement( Node::x ) << '\t';
out << n[i]->displacement( Node::y ) << '\t';
```

As a final description of the results, a table showing element axial forces in produced. Once again it is quite simply done aside from the formatting statements:

```
out << t[i]->axialForce( d ) << '\n';
```

An Example Problem Solved

In order to demonstrate how the library performs, an example problem will be shown. The example truss model is shown in figure 9. The complete input and output files may be found in the appendices.

The results of the analysis are summarized in figure 10, which shows the axial forces (positive tension, negative compression) in the elements. In addition, nodal displacement data was used to create the displaced position sketch.

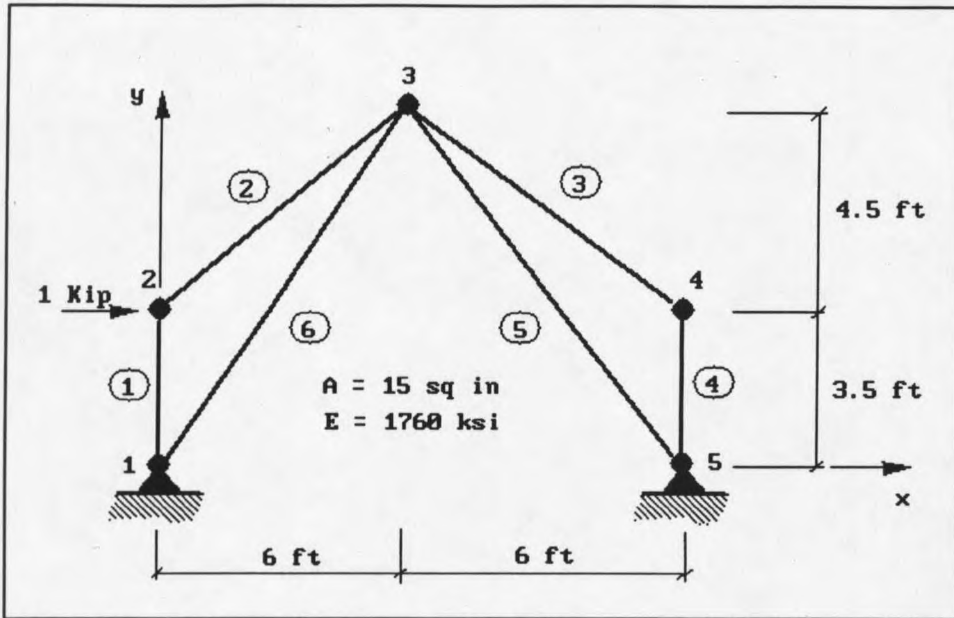

Figure 9: Example truss model

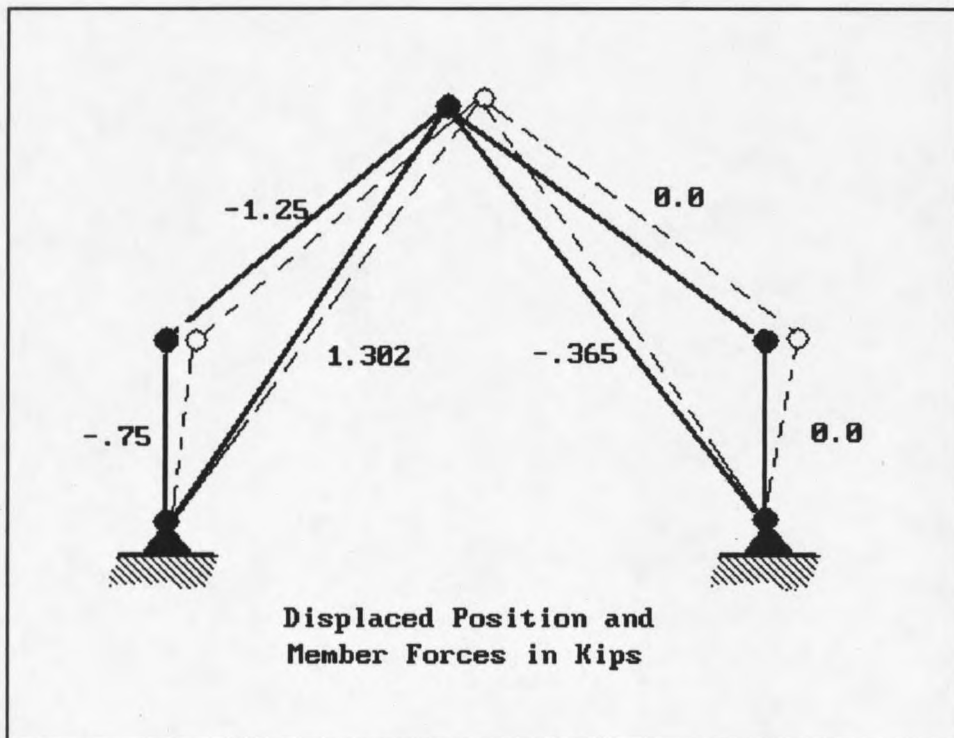

Figure 10: Example truss results

CHAPTER 6

CONCLUSIONS

Summary

This paper has shown the design and implementation of an object oriented library of classes for finite element analysis. The design of the library was restricted to static linear analysis of structures. This library was not intended to completely cover the finite element method--indeed, only the very basic concepts have been represented. However, the design was created with an eye toward future expansion of element and problem types.

The implementation was done in C++, and an exemplar system was created to demonstrate the workings and the potential of the library. C++ is recognized as having many attributes which make it suitable for this highly numerical engineering method. It was intended that the software would be enhanced in such a way as to make it resemble more closely the treatment of finite element analysis as it is generally presented. Each concept in the analysis procedure has been shown to have a corresponding class in the library. This abstraction technique should make modifications and maintenance of the library significantly easier than a functionally similar FORTRAN implementation of the method. If a superior data structure is desired, or a new algorithm becomes available, changes may be made within the appropriate class while no other classes or modules need be aware of such changes.

It was also shown how such an organization leads to a relatively easy implementation. Using a high degree of modularization and encapsulation helped to make the incremental compilation and testing process go quite smoothly. The ability to build on existing concepts as

well as abstract base classes allowed for a simplified derivation of new types as was shown with the Material classes.

Deficiencies in the Library

While C++ has been shown to fulfill many of the expectations in this purpose, it has not proven to be entirely satisfactory. The process of converting from design to program is still quite a significant undertaking. Many of the ideas can become obscured in syntax and the language brings in a measure of complexity along with its power of expression. For example, the idea of operator overloading is quite appealing in its use; yet at the same time is quite tedious to implement.

Another drawback in this system comes from the use of the Rogue Wave classes for matrices, vectors and linear algebra. While these professionally built classes perform efficiently and indeed are quite simple to use, they were not developed as base classes for a derivation hierarchy. Thus, the derivation of a symmetric matrix within the library was not entirely satisfactory. This is because the class DGEMatrix was not designed with virtual functions. Ideally a DSymMatrix should be derived from a general matrix class, so that any function accepting a matrix as a parameter can also accept a symmetric matrix. Within the library this was accomplished with an inferior conversion function built into the symmetric matrix class.

One further change is necessary within the library before it may plausibly be used for the basis of a complete professional level system. The base classes Point and Node have been designed for two dimensional coordinates. Changes in the interface of these classes to allow three dimensional problems to be handled more transparently is recommended.

Possible Extensions

The base library of FEA classes works quite well for the simple truss example program; however with the addition of more complicated element types, and more realistically sized engineering problems to solve, it falls short. Specifically, there is a need for additional support classes to handle elements with complex shape functions. A class representing polynomial functions, a matrix class that could have polynomial functions as elements, and a class to represent numerical integration are envisioned. Such classes would greatly enhance the library, making derivation of this type of element significantly easier and less error prone.

Also of interest would be an object oriented set of utility programs to go along with the library. These could handle the user interface, graphics support and other processing tasks common to finite element programs.

As a final note, C++ is still a currently evolving language. During the development of this thesis, new features have been added to the language which have the potential to improve upon this implementation. Templates provide in C++ an ability to have parameterized types, thus instead of having separate classes for vectors of integers, doubles, etc., one template class can be created which will allow instantiation of any of these. The other significant addition to the language has been exception handling which could be used to provide a more complete and responsive error reporting and control system.

REFERENCES CITED

REFERENCES CITED

[Abelson 85] Abelson, Harold and Sussman, Gerald Jay, Structure and Interpretation of Computer Programs, The MIT Press, 1985.

[Baugh 91] Baugh, John W. and Rehak, Daniel R., "Data Abstraction in Engineering Software Development", Journal of Computing in Civil Engineering, American Society of Civil Engineers, 1991.

[Baugh 89A] Baugh, John W., "Computational Abstractions for Finite Element Programming", Technical Report R-89-182, Department of Civil Engineering, Carnegie Mellon University, 1989.

[Baugh 89B] Baugh, John W. and Rehak, Daniel R., "Object-Oriented Design of Finite Element Programs", Computer Utilization in Structural Engineering, Proceedings from Structures Congress 89. Editor: James K. Nelson Jr. ASCE, NY, NY. 1989.

[Blair 90] Blair, Gordon (ed. et al), Object-Oriented Languages: systems and Applications, Halsted Press (John Wiley & Sons), 1990.

[Booch 91] Booch, Grady, Object Oriented Design with Applications, Benjamin/Cummings, 1991.

[Clough 89] Clough, Ray W., "Original Formulation of the Finite Element Method", Computer Utilization in Structural Engineering, Proceedings from Structures Congress 89. Editor: James K. Nelson Jr. ASCE, NY, NY. 1989.

[Coplien 92] Coplien, James O., Advanced C++ Programming Styles and Idioms, Addison-Wesley, 1992

[Cox 86] Cox, Brad J., Object Oriented Programming An Evolutionary Approach, Addison-Wesley, 1986.

[Fenves 89] Fenves, Gregory L., "Object Oriented Models for Engineering Data", Computing in Civil Engineering (Computers in Engineering Practice), Edited by Thomas O. Barnwell, Jr. ASCE, NY, NY. 1989.

[Horstmann 91] Horstmann, Cay S., Mastering C++: An Introduction to C++ and Object-oriented Programming for C and Pascal Programmers, John Wiley & Sons. 1991.

[Jordan 90] Jordan, David, "Implementation Benefits of C++ Language Mechanisms", Communications of the ACM, Association for Computing Machinery, September 1990.

[Laible 85] Laible, Jeffrey P., Structural Analysis, Holt, Rinehart, Winston. 1985.

[Nagy 78] Nagy, Dennis A., "Software Engineering for Finite Element Analysis", Journal of the Structural Division, American Society of Civil Engineers, August 1978.

[Rogue Wave 91] Math.h++ Class Library, version 3.5, Rogue Wave Associates, Corvallis, OR, 1991.

[Stroustrup 91] Stroustrup, Bjarne, The C++ Programming Language, Second Edition, Addison-Wesley, 1991.

[Weaver 84] Weaver, William Jr. and Johnston, Paul R., Finite Elements for Structural Analysis, Prentice-Hall, 1984.

[Wiener 88] Wiener, Richard S. and Pinson, Lewis J., An Introduction to Object-Oriented Programming and C++, Addison-Wesley, 1988.

[Zienkiewicz 89] Zienkiewicz, O. C. and Taylor, R. L. The Finite Element Mathod--4th ed., Vol. 1: Basic Formulation an Linear Problems, McGraw-Hill, 1989.

APPENDICES

APPENDIX A

C++ LIBRARY HEADER FILES

Figure 11: CoordSys.h Interface for class CoordSystem

```
// TeK  4/27/92

#if ! defined( COORDSYSTEM_H )
#define COORDSYSTEM_H

#if !defined( GLOBAL_H )
#   include <global.h>
#endif

#include <Point.h>

class CoordSystem
{
public:
// destructor
   virtual ~CoordSystem() {}
// constructors
            CoordSystem();
            CoordSystem( double x, double y, double theta );
            CoordSystem( const Point &p1, double theta );
            CoordSystem( double x1, double y1, double x2, double y2 );
            CoordSystem( const Point &p1, const Point &p2 );
            CoordSystem( const CoordSystem &c );

// selectors
   double              angle() const { return thetaX; }
   DGEMatrix           transpose() const { return ::transpose( R ); }
   DGEMatrix           orientation() const { return R; }
   virtual Point       origin() const { return ref; }
   virtual Point       mapToGlobal( const Point &p ) const;
   virtual Point       mapToLocal( const Point &p ) const;
   virtual void        printOn( ostream &s ) const;

// manipulator
   virtual CoordSystem& operator = ( const CoordSystem &c );

private:
   Point ref;
   double thetaX;
   DGEMatrix R;
};  // end class CoordSystem

// related global functions
ostream& operator << ( ostream &s, const CoordSystem &c );

// inline implementations
inline ostream& operator << ( ostream &s, const CoordSystem &c )
{
   c.printOn( s );
   return s;
}

#endif // ifndef COORDSYSTEM_H
```

Figure 12: Dof.h Interface for class Dof (Degree of Freedom)

```
// TeK  4/27/92

#if ! defined( DOF_H )
#define DOF_H

#if !defined( GLOBAL_H )
#   include <global.h>
#endif

class Dof
{
public:
   enum state { free, fixed, prescribed };
   // constructors
                    Dof() : id(-1), status(free), action(0) {}
                    Dof( int i, state f );
                    Dof( const Dof &d )
                       : id(d.id), status(d.status), action(d.action) {}
   // selectors
   int              idNumber() const { return id; }
   boolean          isFixed() const;
   boolean          isFree() const;
   boolean          isPrescribed() const;
   double           displacementValue() const;
   double           loadValue() const;
   void             assemble( DGEMatrix &K, DoubleVec &F ) const;
   void             printOn( ostream &s ) const;

   // modifiers
   Dof&             operator = ( const Dof &d );
   void             addLoad( double load );
   void             setDisplacement( double displacement = 0.0 );
   void             reset();

private:
   int id;
   state status;
   double action;
};  // end class Dof

// related global functions
ostream& operator << ( ostream &s, const Dof &d );

// inline implementations
inline boolean Dof::isFixed() const {
   return (status == fixed) ? TRUE : FALSE;
}

inline boolean Dof::isFree() const {
   return (status == free) ? TRUE : FALSE;
}

inline boolean Dof::isPrescribed() const {
   return (status == prescribed) ? TRUE : FALSE;
}
```

```
inline double Dof::displacementValue() const {
   if( status == free ) {
      cerr << "\nError: Dof::displacementValue()\n";
      cerr << "dof #" << id << " is free.\n";
      exit( -1 );
   }
   else return action;
}

inline double Dof::loadValue() const {
   if( status != free ) {
      cerr << "\nError: Dof::loadValue()\n";
      cerr << "dof #" << id << " is fixed or prescribed.\n";
      exit( -1 );
   }
   else return action;
}

inline ostream& operator << ( ostream &s, const Dof &d ) {
   d.printOn( s );
   return s;
}

#endif // ifndef DOF_H
```

Figure 13: DSymMat.h Interface for class DSymMatrix

```
// TeK  6/11/92

#if ! defined( DSYMMATRIX_H )
#define DSYMMATRIX_H

#if !defined( GLOBAL_H )
#   include <global.h>
#endif

class DSymMatrix : public DoubleVec {
public:
// constructors
               DSymMatrix() : DoubleVec(), squareSize(0) {}
               DSymMatrix( const DSymMatrix &sm );
               DSymMatrix( unsigned size, double values = 0.0 );
               DSymMatrix( const DGEMatrix &m, boolean lower = FALSE );
// conversion function
               operator DGEMatrix() const;

// selectors
   double&        operator() ( unsigned i, unsigned j ) const;
   DoubleVec      operator[] ( unsigned j ) const;
   DoubleVec      row( unsigned i ) const { return (*this)[i]; }
   DoubleVec      col( unsigned j ) const { return (*this)[j]; }
   DGEMatrix      product( const DGEMatrix &m ) const;
   DoubleVec      product( const DoubleVec &vec ) const;
   unsigned       rows() const { return squareSize; }
   unsigned       cols() const { return squareSize; }
   DSymMatrix     copy() const;
   void           printOn( ostream &s ) const;

// modifiers
   void           resize( unsigned size );
   DSymMatrix&    operator = ( const DSymMatrix &sm );

private:
   unsigned       squareSize;
};  // end class DSymMatrix

// other related functions
ostream&          operator << ( ostream &s, const DSymMatrix &sm );
DSymMatrix        transpose( const DSymMatrix &sm );
boolean           isSymmetric( const DSymMatrix &sm );
boolean           isSymmetric( const DGEMatrix &m );

// Inline implementations
inline ostream& operator << ( ostream &s, const DSymMatrix &sm ) {
   sm.printOn( s );
   return s;
}

inline DSymMatrix transpose( const DSymMatrix &sm ) { return sm; };

inline boolean isSymmetric( const DSymMatrix &sm ) { return TRUE; }

#endif   // DSymMat.h
```

Figure 14: Element.h Interface for class Element

```
// TeK  6/3/92

#if ! defined( ELEMENT_H )
#define ELEMENT_H

#if !defined( GLOBAL_H )
#   include <global.h>
#endif

#include "Node.h"
#include "CoordSys.h"
#include "Material.h"

class Element   // Abstract
{
public:
   enum orientation { local, global, principle };
   enum where { atDefault, atNodes, atGauss, atCentroid, anAverage,
                theMaximum, theMinimum };
// destructor
   virtual              ~Element() {}
// constructors
                         Element();
                         Element( const Element &e );
                         Element( int uid, int nc );
// selectors
   int                  idNumber() const { return id; }
   int                  numNodes() const { return nn; }
   CoordSystem&         localCoord() const { return L; }
   virtual void         printOn( ostream &s ) const;
   virtual DoubleVec    findStresses( DoubleVec d, orientation axis =
                     local, where r = atDefault ) const = 0;
   virtual DoubleVec    findStrains( DoubleVec d, orientation axis =
                    local, where r = atDefault ) const = 0;
   virtual DGEMatrix    localStiffness() const = 0;

// modifiers
   virtual Element&     operator = ( const Element &e );
   virtual void         assembleStiffness( DGEMatrix &K ) = 0;

protected:
   CoordSystem L;
   DGEMatrix k;
private:
   int id;
   int nn;
};  // end class Element

// related global functions
ostream& operator << ( ostream &s, const Element &e );

// inline implementations
inline ostream& operator << ( ostream &s, const Element &e )
   {  e.printOn( s );  return s;  }

#endif // ifndef ELEMENT_H
```

Figure 15: FEM.H Include file for Finite Element Method classes

```
// TeK  6/5/92            (link with FEM.LIB)

#include <global.h>

#include<point.h>
#include<coordsys.h>
#include<dof.h>
#include<dsymmat.h>
#include<node.h>
#include<trusnode.h>
#include<material.h>
#include<isotrop.h>
#include<element.h>
#include<truselem.h>
```

Figure 16: global.h

```
// TeK  6/11/92

#if !defined( GLOBAL_H )
#define GLOBAL_H

#if !defined( __STDIO_H )
#   include <stdio.h>
#endif

#if !defined( __IOSTREAM_H )
#   include <iostream.h>
#endif

#if !defined( __IOMANIP_H )
#   include <iomanip.h>
#endif

#if !defined( __STDLIB_H )
#   include <stdlib.h>
#endif

#if !defined( __MATH_H )
#   include <math.h>
#endif

#if !defined( __STRING_H )
#   include <string.h>
#endif

#if !defined( __INTVEC_H__ )
#   include <ivec.h>
#endif

#if !defined( __DOUBLEVEC_H__ )
#   include <dvec.h>
#endif

#if !defined( __DGEMATRIX_H__ )
#   include <dgemat.h>
#endif

#if !defined( __DLUDECMP_H__ )
#   include <dludecmp.h>
#endif

typedef unsigned boolean;

#define TRUE 1
#define true 1
#define FALSE 0
#define false 0
#define TOLERANCE  0.01

#endif // GLOBAL_H
```

Figure 17: Isotrop.h Interface for class Isotropic

```
// TeK 4/23/92, modified 6/11/92

#if ! defined( ISOTROPIC_H )
#define ISOTROPIC_H

#include "Material.h"

class Isotropic : public Material
{
public:
// destructor
   virtual          ~Isotropic() {}

// constructors
                    Isotropic();
                    Isotropic( int idNumber, char *name,
                               spacial dimension,
                               double e = 1.0, double nu = 0.0 );
                    Isotropic( const Isotropic &m );

// selectors
   virtual unsigned   eSize() const { return E.rows(); }
   virtual double     youngsModulus( unsigned direction = 1) const
                         { return ym; }
   virtual double     poissonsRatio( unsigned direction = 1) const
                         { return v; }
   virtual void       printOn( ostream &out, boolean debug = FALSE )
                        const;
   virtual DGEMatrix  e( double theta = 0, double phi = 0) const;
   virtual DGEMatrix  c( double theta = 0, double phi = 0) const;

// modifiers
   virtual Isotropic& operator = ( const Isotropic &iso );

private:
   DSymMatrix E;
   double ym, v;
};  // end class Isotropic


// inline implementations
inline void Isotropic::printOn( ostream &out, boolean debug ) const {
   out << "Isotropic ";
   Material::printOn( out );
   if( debug ) {
      out << "e = " << ym << " nu = " << v << endl;
   }
   out << "[E] is " << E << endl;
   return;
}

#endif // ifndef ISOTROPIC_H
```

Figure 18: Material.h Interface for class Material (Abstract)

```
// TeK  4/23/92, modified 6/11/92

#if ! defined( MATERIAL_H )
#define MATERIAL_H

#if !defined( GLOBAL_H )
#   include <global.h>
#endif

#include <dsymmat.h>

class Material  // Abstract
{
public:
// type field
enum spacial { scalar, planeStress, planeStrain, axisymmetric, solid };
// destructor
   virtual              ~Material() { delete label; }

// modifiers
   virtual Material&   operator = ( const Material &m );

// selectors
   const char const*   name() const { return label; }
   int                 idNumber() const { return id; }
   virtual unsigned    eSize() const = 0;
   virtual void        printOn( ostream &out, boolean debug = FALSE )
                        const;
   virtual double      youngsModulus( unsigned direction = 1 ) const = 0;
   virtual double      poissonsRatio( unsigned direction = 1 ) const = 0;
   virtual DGEMatrix   e( double theta = 0.0, double phi = 0.0 )
                          const = 0;
   virtual DGEMatrix   c( double theta = 0.0, double phi = 0.0 )
                          const = 0;
protected:
// constructors
   Material();
   Material( int idNumber, char *name, spacial dimension );
   Material( const Material &m );
   spacial    d;
private:
   int        id;
   char       *label;
};  // end class Material

// related global functions
ostream& operator << ( ostream &s, const Material &m );

// inline implementations
inline void Material::printOn( ostream &out, boolean debug ) const
{ out << "Material #" << (id+1) << ": " << label << endl; }

inline ostream& operator << ( ostream &s, const Material &m )
   { m.printOn( s ); return s; }

#endif // ifndef MATERIAL_H
```

Figure 19: Node.h Interface for class Node (abstract)

```
// TeK  4/29/92

#if ! defined( NODE_H )
#define NODE_H

#if !defined( GLOBAL_H )
#   include <global.h>
#endif

#include "Point.h"
#include <ivec.h>

class Node : public Point
{
public:
// flags
   enum direction { x, y, z, dx, dy, dz };
// destructor
   virtual   ~Node() {}
// constructors
               Node();
               Node( int idNumber, double x, double y, int numDof = 2 );
               Node( int idNumber, const Point &p, int numDof = 2 );
               Node( const Node &n );
// selectors
   int            idNumber() const { return id; }
   int            numElements() const { return elemIDs.length(); }
   int            numDofs() const { return nd; }
   int            idElement( int i ) const { return elemIDs[i]; }
   virtual void   assemble( DGEMatrix &K, DoubleVec &F ) const = 0;
   virtual double load( direction axis ) const = 0;
   virtual double displacement( direction axis ) const = 0;
   virtual void   printOn( ostream &s ) const;
// modifiers
   void           connectElement( int elementID );
   Node&          operator = ( const Node &n );
   virtual void   disassemble( const DoubleVec &D ) = 0;
   virtual void   makeSupport( direction axis ) = 0;
   virtual void   applyLoad( direction axis, double load ) = 0;
   virtual void   applyDisplacement( direction axis, double d = 0 ) = 0;
   virtual void   clearLoads() = 0;
protected:
   int id;                        // Identification number for node
   IntVec elemIDs;                // Connected element id #s
   int nd;                        // number of degrees of freedom
};  // end class Node

// related global functions
ostream& operator << ( ostream &s, const Node &n );

// inline implementations
inline ostream& operator << ( ostream &s, const Node &n )
{  n.printOn( s );  return s; }

#endif // ifndef NODE_H
```

Figure 20: Ortho.h Interface for class Orthotropic

```
// TeK  6/11/92

#if ! defined( ORTHOTROPIC_H )
#define ORTHOTROPIC_H

#include "Material.h"

class Orthotropic : public Material
{
public:
// destructor
   virtual   ~Orthotropic() {}

// constructors
                Orthotropic();
                Orthotropic( int idNumber, char *name, spacial dimension,
                             double ex, double ey, double nxy, double nyx,
                             double nxz = 0.0, double nzx = 0.0,
                             double nyz = 0.0, double nzy = 0.0 );
                Orthotropic( const Orthotropic &m );

// selectors
   virtual unsigned   eSize() const { return E.rows(); }
   virtual double     youngsModulus( unsigned direction ) const;
   virtual double     poissonsRatio( unsigned direction ) const;
   virtual void       printOn( ostream &out, boolean debug ) const;
   virtual DGEMatrix  e( double theta = 0.0, double phi = 0.0 ) const;
   virtual DGEMatrix  c( double theta = 0.0, double phi = 0.0 ) const;

// modifiers
   virtual Orthotropic& operator = ( const Orthotropic &otm );

private:
   DGEMatrix E;
   double ymx, ymy, vxy, vxz, vyx, vyz, vzx, vzy;
};  // end class Orthotropic


// inline implementations
inline void Orthotropic::printOn( ostream &out, boolean debug ) const {
   out << "Orthotropic ";
   Material::printOn( out );
   if( debug ) {
      out << "Ex = " << ymx << " Ey = " << ymy << endl;
      out << "Vxy = " << vxy << " Vyx = " << vyx << endl;
   }
   out << "[E] is " << E << endl;
   return;
}

#endif // ifndef ORTHOTROPIC_H
```

Figure 21: Point.h Interface for class Point

```
//   TeK  4/20/92

#if ! defined( POINT_H )
#define POINT_H

#include <iostream.h>

class Point
{
public:
// constructors
                      Point() : xc(0.0), yc(0.0) {}
                      Point( double x, double y = 0.0 ) : xc(x), yc(y) {}
// selectors
   double             x() const;
   double             y() const;
   virtual double     distanceTo( const Point &p ) const;
   virtual void       printOn( ostream &s ) const;
   virtual Point      midPoint( const Point &p ) const;
   virtual Point      operator + ( const Point &p ) const;
   virtual Point      operator - ( const Point &p ) const;
// modifiers
   virtual Point&     operator = ( const Point &p );
   virtual void       readFrom( istream &s );

protected:
   double xc;
   double yc;
};  // end class Point

// Related global functions
inline ostream& operator<< ( ostream &s, const Point &p )
    { p.printOn( s ); return s; }
inline istream& operator>> ( istream &s, Point &p )
    { p.readFrom( s ); return s; }

// Inline implementations
inline double Point::x() const
   { return xc; }
inline double Point::y() const
   { return yc; }
inline Point& Point::operator = ( const Point &p )
   { xc = p.x(); yc = p.y(); return *this; }

#endif  // ifndef POINTS_H
```

Figure 22: TrusElem.h Interface for class TrussElement

```
// TeK  6/3/92

#if ! defined( TRUSSELEMENT_H )
#define TRUSSELEMENT_H

#include "Element.h"
#include "TrusNode.h"
#include "Isotrop.h"

class TrussElement : public Element
{
public:
// destructor
   virtual              ~TrussElement() {}
// constructors
                         TrussElement();
                         TrussElement( const TrussElement &te );
                         TrussElement( int uid, Isotropic &mat,
                                       TrussNode &n1, TrussNode &n2,
                                       double crossArea );

// selectors
   double               area() const { return a; }
   double               length() const { return l; }
   Isotropic            material() const { return m; }
   double               axialForce( const DoubleVec &d ) const;
   double               axialStress( const DoubleVec &d ) const;
   double               axialStrain( const DoubleVec &d ) const;
   virtual void         printOn( ostream &s ) const;
   virtual DoubleVec    findStresses( DoubleVec d, orientation axis=local,
                                      where r = atDefault ) const;
   virtual DoubleVec    findStrains( DoubleVec d, orientation axis=local,
                                     where r = atDefault ) const;
   virtual DGEMatrix    localStiffness() const;

// modifiers
   virtual TrussElement&  operator = ( const TrussElement &te );
   virtual void           assembleStiffness( DGEMatrix &K );

private:
   TrussNode *node[2];
   Isotropic &m;
   double a;
   double l;
};  // end class TrussElement

#endif // ifndef TRUSSELEMENT_H
```

Figure 23: TrusNode.h Interface for class TrussNode

```
// TeK  4/29/92

#if ! defined( TRUSSNODE_H )
#define TRUSSNODE_H

#include "Node.h"
#include "Dof.h"

class TrussNode : public Node
{
public:
// destructor
   virtual         ~TrussNode() {}
// constructors
                    TrussNode();
                    TrussNode( const TrussNode & );
                    TrussNode( int idNumber, double x, double y );
                    TrussNode( int idNumber, const Point &p );

// selectors
   virtual void     assemble( DGEMatrix &K, DoubleVec &F ) const;
   virtual double   load( direction axis ) const;
   virtual double   displacement( direction axis ) const;
   virtual void     printOn( ostream &s ) const;

// modifiers
   virtual TrussNode&
                    operator = ( const TrussNode &n );
   virtual void     disassemble( const DoubleVec &D );
   virtual void     makeSupport( direction axis );
   virtual void     applyLoad( direction axis, double load );
   virtual void     applyDisplacement( direction axis, double disp );
   virtual void     clearLoads();

private:
// internal use only
   int              getDofNumber( direction axis, char *fname ) const;

private:
   Dof        dofs[2];
   boolean    haveResults;
   double     results[2];

};  // end class TrussNode

// related global functions
 ostream& operator << ( ostream &s, const TrussNode &tn );


// inline implementations
inline ostream& operator << ( ostream &s, const TrussNode &tn ) {
tn.printOn(s);
   return s;
}

#endif // ifndef TRUSSNODE_H
```

APPENDIX B

INPUT AND OUTPUT FILES FOR EXAMPLE

Figure 24: Input Data

```
* Test Truss #5:  EXAMPLE FROM TEXT OF THESIS
* Units are Kips and Inches
* TeK 7/8/92
Structure: Example 9.5 ([Laible 85] p.656)
     Dimensions 2
     Elements 6
     Nodes 5
     Supports 4
     Materials 1
     Load Cases 1
Coordinates
     1   0.0    0.0
     2   0.0   42.0
     3  72.0   96.0
     4 144.0   42.0
     5 144.0    0.0
Supports
     1   X
     1   Y
     5   X
     5   Y
Materials
     1  Type1  1760.0
Elements
     1  1  2  1  15.0
     2  2  3  1  15.0
     3  3  4  1  15.0
     4  4  5  1  15.0
     5  3  5  1  15.0
     6  1  3  1  15.0
Load Case 1
     2  Force  X  1.0
End Load Case
End
```

Figure 25: Output Data

```
EXEMPLAR:  TRUSS FEM PROGRAM        TeK  June 1992

TRUSS DATA: EXAMPLE FROM TEXT OF THESIS
===========

NODES:
TrussNode #1 at ( 0, 0 )    connected elements: 0 5
   has 2 degrees of freedom
   Degree of Freedom #1 is fixed.
   Degree of Freedom #2 is fixed.

TrussNode #2 at ( 0, 42 )    connected elements: 0 1
   has 2 degrees of freedom
   Degree of Freedom #3 is free.
   Degree of Freedom #4 is free.

TrussNode #3 at ( 72, 96 )    connected elements: 1 2 4 5
   has 2 degrees of freedom
   Degree of Freedom #5 is free.
   Degree of Freedom #6 is free.

TrussNode #4 at ( 144, 42 )    connected elements: 2 3
   has 2 degrees of freedom
   Degree of Freedom #7 is free.
   Degree of Freedom #8 is free.

TrussNode #5 at ( 144, 0 )    connected elements: 3 4
   has 2 degrees of freedom
   Degree of Freedom #9 is fixed.
   Degree of Freedom #10 is fixed.


MATERIALS:
Isotropic Material #1: Type1
[E] is Symmetric Matrix: 1x1
[
    1.76e+03
]
```

```
ELEMENTS:
TrussElement #1:
   Local Coordinate System: at (0, 0), oriented at 90°
   nodes are #1 and #2
   Material: Type1, E = 1.76e+03
   length = 42, area = 15

TrussElement #2:
   Local Coordinate System: at (0, 42), oriented at 36.87°
   nodes are #2 and #3
   Material: Type1, E = 1.76e+03
   length = 90, area = 15

TrussElement #3:
   Local Coordinate System: at (72, 96), oriented at -36.87°
   nodes are #3 and #4
   Material: Type1, E = 1.76e+03
   length = 90, area = 15

TrussElement #4:
   Local Coordinate System: at (144, 42), oriented at -90°
   nodes are #4 and #5
   Material: Type1, E = 1.76e+03
   length = 42, area = 15

TrussElement #5:
   Local Coordinate System: at (72, 96), oriented at -53.13°
   nodes are #3 and #5
   Material: Type1, E = 1.76e+03
   length = 120, area = 15

TrussElement #6:
   Local Coordinate System: at (0, 0), oriented at 53.13°
   nodes are #1 and #3
   Material: Type1, E = 1.76e+03
   length = 120, area = 15
```

```
LOAD CASE #1
============
JOINT LOADS AND DISPLACEMENTS
TrussNode #1 at ( 0, 0 )    connected elements: 0 5
   has 2 degrees of freedom
   Degree of Freedom #1 is fixed.
   Degree of Freedom #2 is fixed.
   Results have been dissasembled, displacements are:
   x: 0  y: 0

TrussNode #2 at ( 0, 42 )    connected elements: 0 1
   has 2 degrees of freedom
   Degree of Freedom #3 is free, applied loads = 1
   Degree of Freedom #4 is free, applied loads = 0
   Results have been dissasembled, displacements are:
   x: 0.014532  y: -0.001193

TrussNode #3 at ( 72, 96 )    connected elements: 1 2 4 5
   has 2 degrees of freedom
   Degree of Freedom #5 is free, applied loads = 0
   Degree of Freedom #6 is free, applied loads = 0
   Results have been dissasembled, displacements are:
   x: 0.006313  y: 0.002663

TrussNode #4 at ( 144, 42 )    connected elements: 2 3
   has 2 degrees of freedom
   Degree of Freedom #7 is free, applied loads = 0
   Degree of Freedom #8 is free, applied loads = 0
   Results have been dissasembled, displacements are:
   x: 0.004316  y: 1.156365e-10

TrussNode #5 at ( 144, 0 )    connected elements: 3 4
   has 2 degrees of freedom
   Degree of Freedom #9 is fixed.
   Degree of Freedom #10 is fixed.
   Results have been dissasembled, displacements are:
   x: 0  y: 0
```

NODAL DISPLACEMENTS

Node	dx	dy
1	0.0000	0.0000
2	0.0145	-0.0012
3	0.0063	0.0027
4	0.0043	0.0000
5	0.0000	0.0000

ELEMENT AXIAL FORCES

Bar	Force
1	-0.7484
2	-1.2499
3	0.0000
4	0.0001
5	-0.3645
6	1.3021

MONTANA STATE UNIVERSITY LIBRARIES
3 1762 10176010 4

HOUCHEN
BINDERY LTD
UTICA/OMAHA