DEVELOPMENT OF A WINDOWS BASED ANALYSIS TOOL FOR STRUCTURAL ANALYSIS AND DYNAMICS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

SONER BAŞ

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE
IN
THE DEPARTMENT OF CIVIL ENGINEERING

APRIL 2003

Approval of the Graduate School of Natural and Applied Sciences

Prof. Dr. Tayfur ÖZTÜRK
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science

Prof. Dr. Mustafa TOKYAY
Head of the Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science

Prof. Dr. Polat GÜLKAN
Supervisor

Examining Committee Members

Prof. Dr. S. Tanvir WASTI

Prof. Dr. Polat GÜLKAN

Prof. Dr. Haluk SUCUOĞLU

Assoc. Prof. Dr. Uğur POLAT

M. Sc. Nejat BAYÜLKE

ABSTRACT

DEVELOPMENT OF A WINDOWS BASED ANALYSIS TOOL FOR STRUCTURAL ANALYSIS AND DYNAMICS

Baş, Soner

M. Sc., Department of Civil Engineering

Supervisor: Prof. Dr. Polat Gülkan

April 2003, 134 pages

At the present time, there are large numbers of computer programs developed for the purpose of static and dynamic analysis of structures. These programs completely automate the analysis process and it is possible for students and users to carry out structural analysis without the understanding of the theory of structural analysis. For such reasons, these programs are not suitable for educational purposes.

In this study, a computer program as an educational tool for structural analysis and dynamics has been developed. The program is based on a former educational software, called CAL91, which was developed by Wilson (1991). The primary goal of the study is that; the program should have a user-friendly interface with visualization support in all three phases of the solution process,

namely pre-processing, analysis and post processing. To achieve this goal, Microsoft Windows operating system was selected as the target platform. The resulting program was named CALWin.

CALWin was developed using C++ programming language using an object-oriented approach. This approach enabled the construction of a modular system for the implementation of static and dynamic analysis procedures, finite elements and graphical objects. This provides a significant advantage for further studies. The OpenGL API was utilized for the development of three-dimensional graphics engine required for the graphic editor.

Finally, three-noded triangular and four-noded quadrilateral plane stress-strain elements, which are not available in CAL91, are added to the finite element library of the program.

A number of benchmark problems have been solved to illustrate the way this software runs.

Keywords: structural analysis, structural dynamics, software, Windows based, object oriented, user-friendly, educational

ÖZ

YAPISAL ANALİZ VE DİNAMİK İÇİN WINDOWS TABANLI BİR ANALİZ YAZILIMININ GELİŞTİRİLMESİ

Baş, Soner

Yüksek Lisans, İnşaat Mühendisliği Bölümü

Tez Yöneticisi: Prof. Dr. Polat Gülkan

Nisan 2003, 134 sayfa

Günümüzde, statik ve dinamik yapı analizi için geliştirilmiş çok sayıda bilgisayar programı mevcuttur. Bu programlar analiz sürecini tamamen otomatik hale getirmiştir. Dolayısıyla kullanıcıların ve öğrencilerin, yapısal analiz teorisini bilmeden de yapısal analizi gerçekleştirmeleri mümkün olmaktadır. Bu nedenle, genel maksatlı programlar eğitim amaçlı kullanıma uygun değildir.

Bu çalışmada, yapısal analiz ve yapısal dinamik için eğitim amaçlı bir bilgisayar programı geliştirilmiştir. Bu programın geliştirilmesinde, Wilson (1991) tarafından yazılan CAL91 isminde eğitim amaçlı başka bir program temel alınmıştır. Bu çalışmanın ana hedefi yazılan programın grafik tabanlı ve kolay kullanımlı bir arabirime sahip olması idi. Program amaca uygun olan Microsoft

Windows platformunda geliştirildi. Bu nedenle programa CALWin ismi verilmiştir.

CALWin, C++ prograiama dili ile nesne yönelimli bir yaklaşım kullanılarak yazılmıştır. Bu yaklaşım, statik ve dinamik işlemleri ile sonlu elemanların ve grafik nesnelerinin bilgisayar kodlamasında modüler bir sistemin kurulmasına olanak sağlamıştır. Böylece daha sonraki çalışamalara da önemli bir avantaj sağlanmıştır. Grafik editör için gerekli olan üç boyutlu grafik motoru OpenGL kullanılarak geliştirilmiştir.

Son olarak, CAL91'de bulunmayan üçgen ve dörtgen düzlemsel gerilme ve düzlemsel birim şekil değiştirme elemanları programa dahil edilmiştir.

Anahtar Kelimeler: yapısal analiz, yapısal dinamik, yazılım, Windows tabanlı, nesne yönelimli, kolay kullanımlı, eğitim amaçlı

To my father

İhsan Baş

ACKNOWLEDGMENTS

First and foremost, I would like to thank to my supervisor Prof. Dr. Polat Gülkan for his support, guidance and patience.

Secondly, I would like to thank to Mr. Ateeq Ahmad for his extreme patience and encouragement. I would also like to thank to my friend Eyyub Çektimur who assisted me in the development of computer program. Thanks are also due to my dear friend Eray Baran for his technical and morale support.

Finally, I express my gratitude to all members of my family for their confidence, encouragement and continual support they have provided in this study and all along my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

FIGURE **PAGE**

CHAPTER 1

INTRODUCTION

Most problems involved in structural analysis and dynamics require heavy computational efforts, like the multiplication, addition and inversion of large matrices and solution of large system of equations. A computer program is always necessary for solving such problems. The aim of the present study is to develop a general-purpose computer program for solving these problems in a practical way through a user-friendly graphical interface. Its end user group is composed of students in an early stage of learning the principles of structural analysis.

A vast amount of computer programs have been developed for the purpose of solving problems arising in structural statics and dynamics. Many of them are for very specific type of problems and some of them are for general-purpose structural analysis. Programs developed for specific type of problems will not be included in the discussion. Among the general purpose programs, SAP2000, LUSAS, ANSYS, STRAND are examples of widely used ones and are very powerful both in terms of analysis capabilities and in terms of ease of use. However, all of these programs are commercial codes and therefore are not suitable for educational purposes, since they mask the details of analysis procedures and do not present commands or tools for performing very simple tasks, like the determination of eigenvalues of a simple standard eigen problem.

For educational purposes, CAL91 has proved to be a very suitable analysis tool, which was first developed by Wilson (1991). In some respects it resembles the programs like MathCAD, MathLab, etc. since it enables users to evaluate very basic operations like matrix multiplications and additions. On the other hand, CAL91 is also able to handle static and dynamic analysis tasks. For example, one can obtain the uncoupled dynamic response of a structural system or can carry out stepwise numerical solution of a dynamic problem. For such reasons it is an appealing educational tool and also has been used in METU as a learning aid for structural analysis and dynamics. However, CAL91 does not have any graphical user interface neither for preprocessing (input) nor for post processing (output). Another drawback of the program is its finite element library. Finite element library of Cal91 does not include general four-noded quadrilateral and three-noded triangular plane elements.

The initial goal of this work is to develop a computer program based on the capabilities of CAL91 with a user-friendly graphical environment while preserving the features that make the CAL91 a comprehensive educational tool. In addition to graphical user interface, the analysis capabilities of CAL91 will be enhanced by adding general quadrilateral and triangular plane stress/strain elements.

Due to the reasons explained in the above paragraphs, the scope of the present study does not include the development of software with fully automated features like the automatic calculation of loads, static and dynamic analysis by clicking a button. It does not include any advanced finite element with orthotropic material features and non-linear capabilities. However, the program has the options for performing static analysis and various types of dynamic analyses

through a graphical interface composed of dialog boxes, drop-down menus, toolbars, tree views and graphical editor windows in which the geometry can be defined visually using the mouse. It is also possible to view member force diagrams and deflected shape of the structure in the post-processing phase.

The programming language chosen for the development of this program is C++. Although C++ was not specifically designed with numerical computation in mind, with the help of Standard Template Library, it is as powerful as Fortran, and much numerical and engineering computation is done in C++ (Stroustrup, 1997) Besides, graphics and user interfaces are areas in which C++ is heavily used. Another reason is that C++ supports object oriented programming.

The target platform is Microsoft Windows. Because this is the most common operating system. The development environment is the Microsoft Visual Studio 6.0. MFC (Microsoft Foundation Classes) constitutes the framework of the GUI (Graphical User Interface). MFC is a library of C++ classes that encapsulates the Windows functions and objects (Blaszczak, 1999). OpenGL is used for the two and three-dimensional visualization of elements. OpenGL is a software interface to graphics hardware that consists of about 250 commands to specify the objects and operations to produce three-dimensional graphics (Woo et al. 1997). The program has been developed using object oriented programming (OOP) approach, which significantly increases the code maintainability, reusability and program modularity.

The developed software has been named as CALWin, because it is based on CAL91 and developed to run in Microsoft Windows OS.

The study is organized as follows: The underlying theory and assumptions of CALWin are presented in Chapters 2,3 and 4. Chapter 2 discusses the principles of matrix structural analysis and emphasizes the role of matrices in analysis procedure. Chapter 3 contains information about the derivation of element matrices of the finite elements implemented. In Chapter 4, dynamic analysis procedures implemented in CALWin are explained briefly. Chapter 5 discusses tools that were used in the development of CALWin and presents an overview of CALWin with the listings of available commands. Structure of the program and class hierarchies are also included in this chapter. A number of benchmark examples are solved in Chapter 6. Chapter 7 presents an overview and summary of the work.

The appendix contains the source code, executable file and setup file of CALWin provided in a CD. The project files of the examples solved in Chapter 6 are also included in the CD.

CHAPTER 2

REVIEW OF PRINCIPLES OF MATRIX STRUCTURAL ANALYSIS

In this chapter, a brief discussion of the principles of matrix structural analysis will be presented. The purpose is not to teach the reader structural analysis but to emphasize the role played by matrix operations in handling a broad variety of problem solving requirements.

In the first part of this chapter, some basic concepts used in the displacement method of matrix structural analysis (with emphasis on the particular form of displacement approach known as the direct stiffness method) are discussed. Next, the equations of displacement based approach are stated. Displacement based method is the preferred method and as in all general-purpose finite element programs it is the method that is implemented and used in the CALWin.

The terms *matrix structural analysis* and *finite element analysis* have similar meanings. Commonly, "matrix analysis" analysis is used to refer the analysis of truss and frame assemblages such as in Figure 2.1, and "finite elements" refers to particular problems of modeling of continua Figure 2.2. The major difference in the solutions of former type of systems when compared to a

more general finite element analysis of two- or three-dimensional problems is that the exact element stiffness matrices ("exact" within limitations of the beam theory) could be calculated. However, it should be obvious that the "matrix analysis" also falls under the heading "finite element analysis".

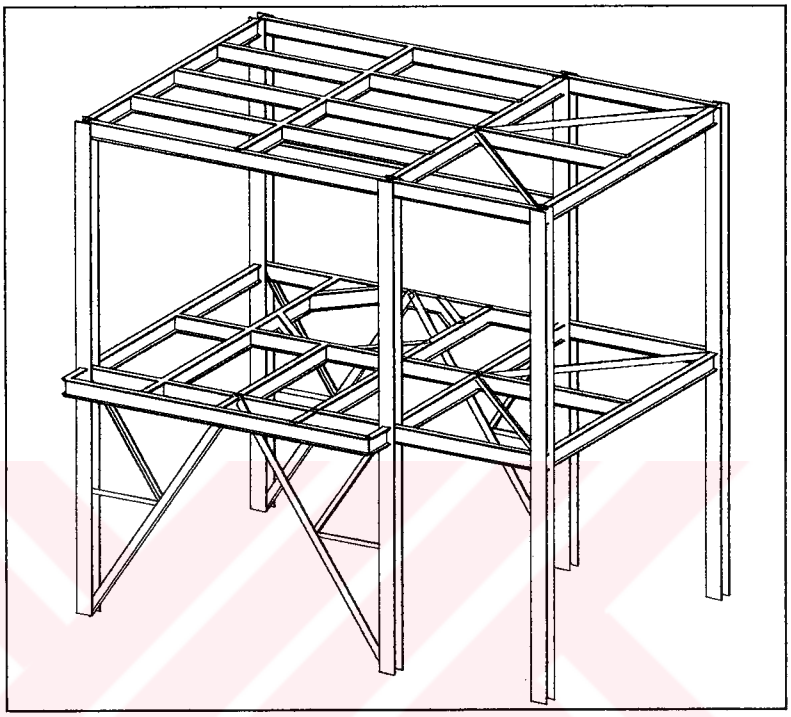


Figure 2.1 A Frame Structure

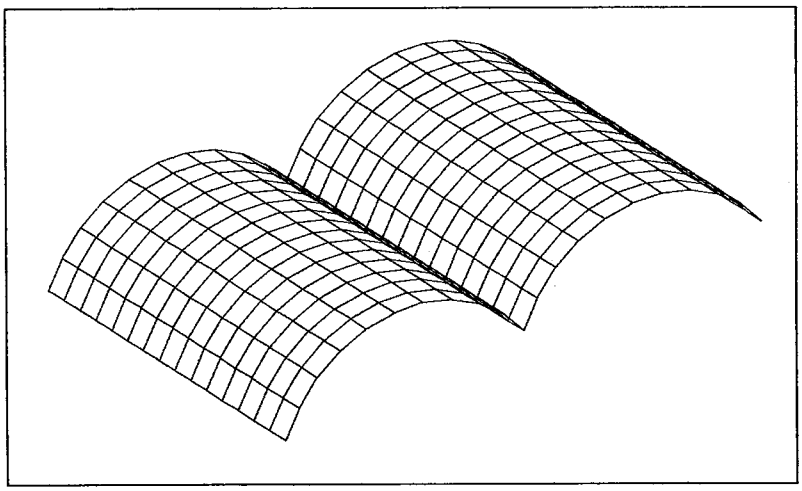


Figure 2.2 A Roof Modeled Using Shell Elements

2.1 Basic Steps of Engineering Analysis

The analysis of an engineering system requires the idealization of the system into a form that can be solved, the formulation of the mathematical model, the solution of this model and the interpretation of results (McGuire and Gallagher, 2000). Viewed in this way, structural analysis may be broken into four parts:

1. *Discretization*. Definition of physical model, geometry, material, loading and boundary conditions.
2. *Mathematical modeling*. Generation of mathematical models for structural elements and applied loadings.
3. *Solution*. The establishment of the governing algebraic equations of the system (mathematical model).
4. *Solution interpretation*. The presentation of results in a form useful in design.

In the discretization phase, structure is divided into a number of physical members. Most commonly used structural elements are shown in Figure 2.3. The first element is a typical framework element and used in analysis of framed structures like the one in Figure 2.1, second one is a shell element and user for modeling structures as shown in Figure 2.2.

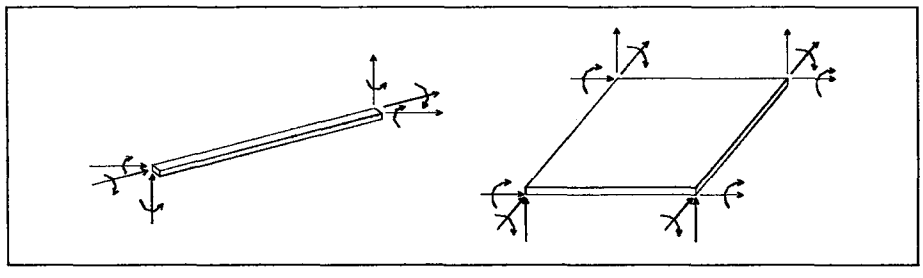


Figure 2.3 Typical Finite Elements for Structural Analysis

The finite element library of the CALWin is the subject of the next chapter. Nevertheless, it is expedient to list here the elements that have been implemented in CALWin. They are displayed in Table 2.1.

Table 2.1 Finite Element Library of CALWin

Element Name	*DOFs per node*
Beam (Slope)	2
Truss	3
Frame 2D	3
Frame 3D	6
Beam on elastic foundation	3
Beam on 2 parameter elastic foundation	3
Pile	3
3-Node triangular plane stress/strain	2
4-Node quadrilateral plane stress/strain	2

In the next phase governing differential equations of the system are formulated at the element level. The basis of formulation is the principle of virtual work, which will be discussed later in this chapter.

Solution phase operates on the equations formed in the prior phase. In the case of a static analysis this may mean no more than a solution of a set of linear system of equations. Solutions for dynamic response may require very expansive computations over a time-history of applied loads. Solution methods for dynamic response are addressed in the Chapter 4. Irrespective of whether static or dynamic analysis is performed, as the total number of equations increases, the effort required for the solution of system of equations increases exponentially. Since it was developed as an educational tool, in the original version of CAL91, there was no efficient equation solver that utilizes the bandedness of a system of equations

generated in the matrix structural analysis method. However, in CALWin, a skyline solver has been implemented.

2.2 Displacement Method of Matrix Structural Analysis

Before the statement of direct stiffness method, a brief explanation of some basic concepts used in the finite element analysis and matrix structural analysis is presented.

2.2.1 Degrees of Freedom

In the displacement method of matrix structural analysis, nodal point displacements are the unknowns to be determined. The displacement components required for definition of the behavior of typical structures are called degrees of freedom. Degrees of freedom for typical structures are shown in the following Figure 2.4.

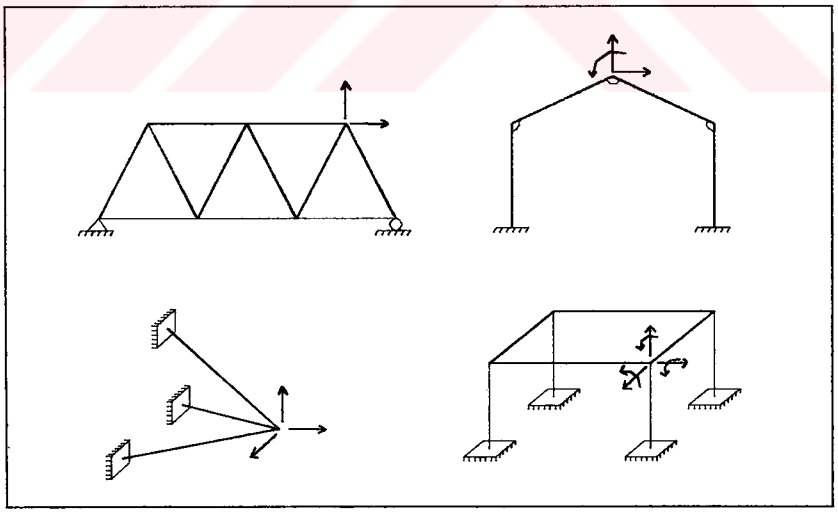


Figure 2.4 Degrees of Freedom for Various Types of Structures

In practice, the number of degrees of freedom of a system is not unique, instead, a function of the manner in which the real structure has been idealized for analysis.

2.2.2 Coordinate Systems

In engineering analysis, right-handed coordinate systems are used to define displacements, forces and material properties.

Direct stiffness method of matrix structural analysis requires the element force-displacement relations are expressed in element basis. Therefore, it is necessary to use a coordinate system specified to each element. Elements' local coordinate system axes will be denoted by a primed lower case letters: x', y', z'.

The solution of equilibrium equations formed per element basis, however, is carried out in a different coordinate system, common to all members. This coordinate system is called global (or structural) coordinate system and will be designated by the symbols: x, y, z as in Figure 2.5.

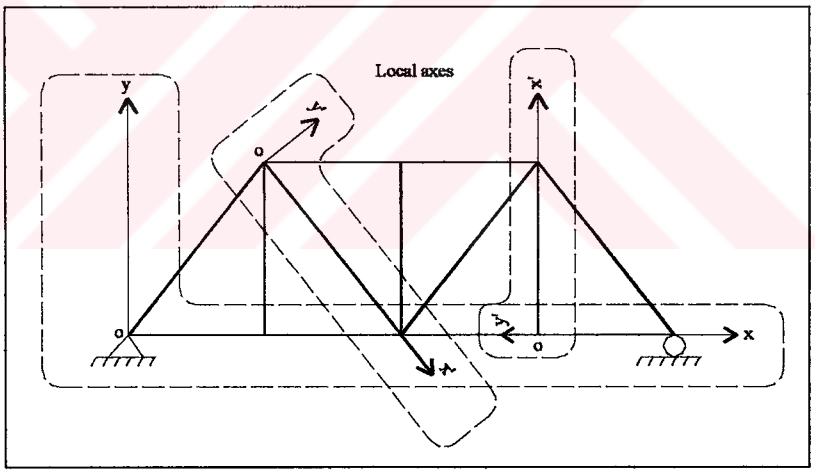


Figure 2.5 Local and Global Axes (McGuire and Gallagher, 2000)

2.2.3 Coordinate Transformations

Transformation of all quantities such as displacements (see Figure 2.6), forces, and force-displacement relations arising in matrix structural analysis from one coordinate system to another are carried out by matrix multiplications. These matrices are called transformation matrices.

In Figure 2.6 displacement components at a node are shown.

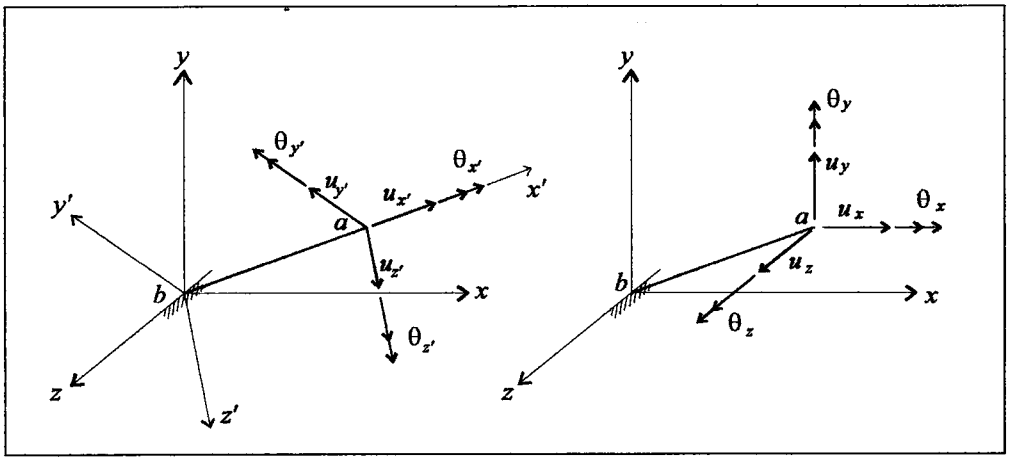


Figure 2.6 Local and Global Components of Displacement of a Node

Following two vectors are the alternative representation of the same quantity:

$$\mathbf{u} = \begin{bmatrix} u_x^1 & u_y^1 & u_z^1 & \theta_x^1 & \theta_y^1 & \theta_z^1 & u_x^2 & u_y^2 & u_z^2 & \theta_x^2 & \theta_y^2 & \theta_z^2 \end{bmatrix}^T$$
$$\mathbf{u'} = \begin{bmatrix} u'^1_x & u'^1_y & u'^1_z & \theta'^1_x & \theta'^1_y & \theta'^1_z & u'^2_x & u'^2_y & u'^2_z & \theta'^2_x & \theta'^2_y & \theta'^2_z \end{bmatrix}^T \quad (2.1)$$

where superscripts denote node number.

Here, **u** is the displacement vector in global coordinates and **u'** is the displacement vector in local coordinates. They are related to each other through:

$$\mathbf{u} = \mathbf{A}\mathbf{u'} \quad (2.2)$$

Here, **A** is the transformation matrix and it is given by:

$$\mathbf{A} = \left[\begin{array}{c|c|c|c} \boldsymbol{\gamma} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \hline \mathbf{0} & \boldsymbol{\gamma} & \mathbf{0} & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} & \boldsymbol{\gamma} & \mathbf{0} \\ \hline \mathbf{0} & \mathbf{0} & \mathbf{0} & \boldsymbol{\gamma} \end{array}\right] \quad (2.3)$$

and γ is the matrix of direction cosines. The γ matrix is defined as:

$$\gamma = \begin{bmatrix} \lambda_{x'} & \mu_{x'} & \upsilon_{x'} \\ \lambda_{y'} & \mu_{y'} & \upsilon_{y'} \\ \lambda_{z'} & \mu_{z'} & \upsilon_{y'} \end{bmatrix} \tag{2.4}$$

Where λ, μ, and δ, with appropriate subscript, designates the cosines of the nine direction angles between the global x, y, and z-axes and the corresponding local axis. They are defined as:

$$\begin{aligned} &\lambda_{x'} = \cos\alpha_{x'}; \quad \mu_{x'} = \cos\beta_{x'}; \quad \upsilon_{x'} = \cos\delta_{x'} \\ &\lambda_{y'} = \cos\alpha_{y'}; \quad \mu_{y'} = \cos\beta_{y'}; \quad \upsilon_{y'} = \cos\delta_{y'} \\ &\lambda_{z'} = \cos\alpha_{z'}; \quad \mu_{z'} = \cos\beta_{z'}; \quad \upsilon_{z'} = \cos\delta_{z'} \end{aligned} \tag{2.5}$$

2.3 Direct Stiffness Method

A general frame element with 6 degrees of freedom per node is shown in Figure 2.7. In matrix structural analysis, relationship between nodal displacements and nodal forces, i.e. force displacement relationships, can be written in three forms:

- Stiffness equations
- Flexibility equations
- Mixed force-displacement equations

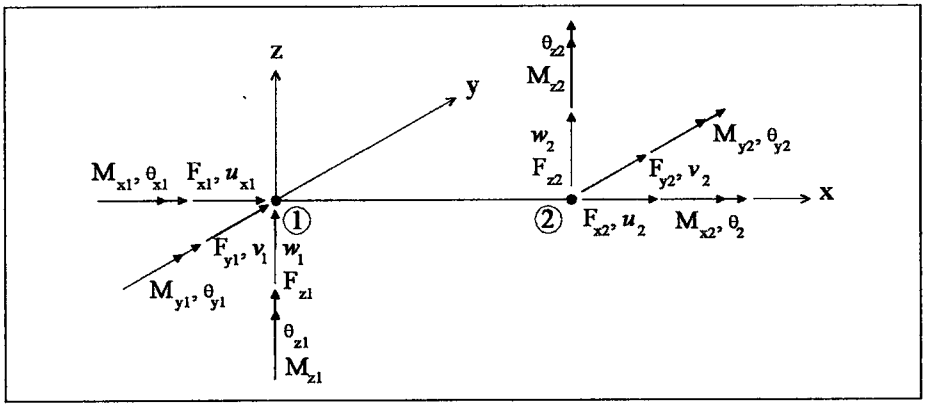


Figure 2.7 Frame Element With 6-Dofs Per Node

Only the first form is discussed in this text, because other two methods do not render well when implemented in computer programs.

As for the stiffness equations, element stiffness equations are linear algebraic equations of the form:

$$\begin{aligned}
f_1' &= k_{11}'u_1' + k_{12}'u_2' + \ldots + k_{1j}'u_j' + \ldots + k_{1n}'u_n' \\
&\vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \\
f_i' &= k_{i1}'u_1' + k_{i2}'u_2' + \ldots + k_{ij}'u_j' + \ldots + k_{in}'u_n' \\
&\vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \\
f_n' &= k_{n1}'u_1' + k_{n2}'u_2' + \ldots + k_{nj}'u_j' + \ldots + k_{nn}'u_n'
\end{aligned} \tag{2.6}$$

or more conveniently, in matrix form:

$$\mathbf{f}' = \mathbf{k}'\mathbf{u}' \tag{2.7}$$

Where, $\mathbf{f}'$ and $\mathbf{u}'$ are the nodal force and displacement vectors respectively. The matrix $\mathbf{k}'$ is the element stiffness matrix. An individual term of the matrix $\mathbf{k}'$, k_{ij}', is called an element stiffness coefficient. Its physical interpretation is as follows: it is the force at degree of freedom i, required to fix the displacement at that degree of freedom to zero when a unit displacement is imposed to degree of

freedom j. More information about the derivation of element stiffness coefficients is given in the next chapter for each element type.

Although we formulate the element force-displacement equations in element coordinates, the solution is related to the complete structural system. Hence, all of these quantities, namely, nodal displacements, nodal forces and stiffness coefficients must be transformed to the global coordinate system. This is accomplished by applying mathematical transformations to above matrices and vectors as described in the previous section.

The transformed set of equations is:

$$\begin{aligned} f_1 &= k_{11}u_1 + k_{12}u_2 + \ldots + k_{1j}u_j + \ldots + k_{1n}u_n \\ &\vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \\ f_i &= k_{i1}u_1 + k_{i2}u_2 + \ldots + k_{ij}u_j + \ldots + k_{in}u_n \\ &\vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \\ f_n &= k_{n1}u_1 + k_{n2}u_2 + \ldots + k_{nj}u_j + \ldots + k_{nn}u_n \end{aligned} \tag{2.8}$$

In matrix form: $$\mathbf{f} = \mathbf{ku} \tag{2.9}$$

Where; $$\begin{aligned} \mathbf{f} &= \mathbf{A}^{\mathrm{T}}\mathbf{f}' \\ \mathbf{k} &= \mathbf{A}^{\mathrm{T}}\mathbf{k}'\mathbf{A} \\ \mathbf{u} &= \mathbf{A}^{\mathrm{T}}\mathbf{u}' \end{aligned} \tag{2.10}$$

f, **k**, **u** are the element force vector, element stiffness matrix and element displacement vector in global coordinates and **A** is the transformation matrix.

In Equation (2.8), the numbers 1...i...n for the degrees of freedom correspond to global numbering system. The global numbering system is completely independent of local element degree of freedom numbering. When the global analysis equations are formed using the direct stiffness method, all degrees of freedom appear in each row of Equation (2.8). However, in general some of

these degrees of freedom are not available due to support conditions. In such cases these degrees of freedom are given the number zero, and do not contribute to the structural stiffness matrix.

Once the element force displacement relations have been numerically evaluated for each member of the structure, application of the direct stiffness method consists of their combination in algebraic form in a manner dictated by the requirements of the static equilibrium and displacement compatibility at the nodal points. This process is called assembly process. An ID matrix is used in the assembly of element stiffness coefficients to the structural stiffness matrix. A row of ID matrix maps the rows and columns of an element stiffness matrix to the structural stiffness matrix.

These operations produce a set of force-displacement equations for the nodal points of the assembled model:

$$\mathbf{F} = \mathbf{KU} \tag{2.11}$$

In this equation **K** is the global stiffness matrix of the whole structure, **U** is the displacement vector and **F** is the structural load vector.

The methods of solution of linear system of equations will not be discussed here. The reader should refer to textbooks on the subject.

When the global set of equations are solved and the unknown structural displacements are determined, element forces are easily obtained through the following matrix multiplications:

$$\mathbf{u}' = \mathbf{Au} \tag{2.12}$$

and

$$\mathbf{f}' = \mathbf{k}'\mathbf{u}' \tag{2.13}$$

Therefore

$$\mathbf{f}' = \mathbf{k}'\mathbf{A}\mathbf{u} \tag{2.14}$$

The element internal forces and stresses which is required for designing the element can then be evaluated using the element nodal point forces.

This completes our discussion of direct stiffness method. As it was stated at the beginning of the chapter, the aim of this text is not to instruct the reader in structural analysis. For this reason, no theoretical explanation was given in this chapter, concerning the derivation of element and structural stiffness equations; however, more detailed procedure of the derivation of element stiffness and as well as mass and geometric stiffness matrices will be given in the next chapter when we present the finite element library of the CALWin program.

As seen from the above discussion, direct stiffness method is a procedure that consists of a series of matrix operations. For that, the basis of all operations in CALWin is the matrix operations. CALWin presents a number of matrix operations, which are summarized in Chapter 5.

CHAPTER 3

FINITE ELEMENT LIBRARY

3.1 One Dimensional Elements

In this section, stiffness matrices of one-dimensional elements that are implemented in CALWin are presented. The stiffness matrices of all one-dimensional elements presented in this section are exact because of the use of exact shape functions.

3.1.1 Truss Element

A truss element is a straight bar connected to other members by momentless joints. All loadings are assumed to be applied only at these points of intersection. Thus a truss element is subjected only to axial force. So, it is the simplest finite element.

A truss element with length L, modulus of elasticity E, and cross sectional area A is shown in Figure 3.1. The bar is subjected to axial loading only.

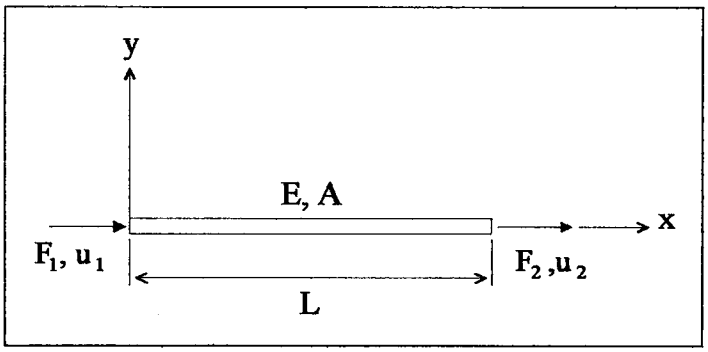


Figure 3.1 Truss Element

The differential equation of equilibrium to be solved is given by:

$$EA\frac{d^2u}{dx^2}+F=0 \tag{3.1}$$

For a bar element with constant axial stress or strain, the axial displacement u(x) at a distance x from node 1 is assumed as varying linearly. Therefore, the displacement u(x) may be written as (Smith and Griffiths, 1998)

$$u(x)=N_1(x)u1+N_2(x)u2 \tag{3.2}$$

where

$$N_1(x)=1-\frac{x}{L} \quad \text{and} \quad N_2(x)=\frac{x}{L} \tag{3.3}$$

therefore;

$$u(x)=\begin{bmatrix}N_1 & N_2\end{bmatrix}\begin{bmatrix}u_1\\u_2\end{bmatrix}=\mathbf{Nu} \tag{3.4}$$

Using the principle of virtual work and Equation (3.4) we obtain the strain energy expression of the bar element as follows:

$$U=\frac{EA}{2}\int_0^L\left(\frac{dN_1}{dx}u_1+\frac{dN_2}{dx}u_2\right)^2dx \tag{3.5}$$

Applying Castigliano's theorem gives

$$F_1 = \left(EA \int_0^L \frac{dN_1}{dx} \frac{dN_1}{dx} dx \right) u_1 + \left(EA \int_0^L \frac{dN_1}{dx} \frac{dN_2}{dx} dx \right) u_2 \tag{3.6a}$$

$$F_2 = \left(EA \int_0^L \frac{dN_2}{dx} \frac{dN_1}{dx} dx \right) u_1 + \left(EA \int_0^L \frac{dN_2}{dx} \frac{dN_2}{dx} dx \right) u_2 \tag{3.6b}$$

or in matrix form:

$$\begin{bmatrix} F_1 \\ F_2 \end{bmatrix} = \begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{3.7}$$

$$\mathbf{F} = \mathbf{ku} \tag{3.8}$$

Here **k** is the stiffness matrix with:

$$k_{ij} = EA \int_0^L \frac{dN_i}{dx} \frac{dN_j}{dx} dx \tag{3.9}$$

Substituting Equation (3.3) into Equation (3.7) and evaluating the integrals we obtain the stiffness matrix as:

$$\mathbf{k} = \frac{EA}{L} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \tag{3.10}$$

The same stiffness matrix is also obtained by applying the Galerkin form of least squares technique to the governing Equation (3.1).

In CALWin the stiffness matrix of a truss element that is oriented arbitrarily in space is calculated. The 6x6 local stiffness matrix of that element is:

$$\mathbf{k} = \frac{EA}{L} \begin{bmatrix} 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.11}$$

However, CALWin always gives the element stiffness matrix in global coordinates so that it can be assembled using direct stiffness method without having the user perform the transformation manually. Positive definition of truss forces and degrees of freedom in CAL is shown in Figure 3.2.

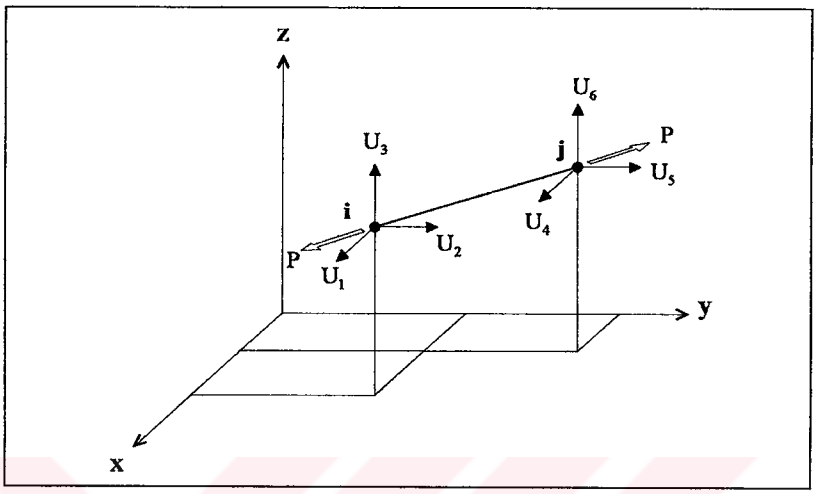


Figure 3.2 Definitions of Positive Displacements and Forces for Truss Eelement

3.1.2 Slope Element

The classical slope-deflection method for horizontal beams and vertical columns is a well-known approach for structural analysis. However, it is identical to the direct stiffness method in which axial deformations are assumed to be zero.

CALWin includes a command for calculating stiffness matrix of linear frame members. The definition of degrees of freedom for this element is given in Figure 3.3.

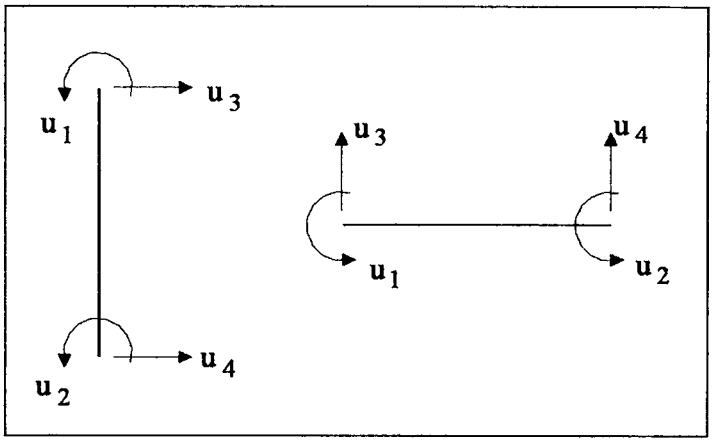


Figure 3.3 Numbering of Dofs of the Slope Element

The stiffness matrix of this element is identical to that of a horizontal beam element except that numbering of degrees of freedom is different. Note; however, that the stiffness matrix of slope element cannot be used for non-horizontal beams and non- vertical columns.

The derivation of stiffness matrix of slope element is discarded since the element stiffness matrix is the same as that of two-dimensional frame element without the axial degrees of freedom.

3.1.3 Two Dimensional Frame Element With Axial Deformations

Two-dimensional frame element can be thought of superposition of a truss element and a beam element. We have already presented the stiffness matrix of truss element and, hence, only the derivation of beam element stiffness will be presented here.

A straight beam element with uniform cross section is shown in Figure 3.4. The element has constant moment of inertia I, modulus of elasticity E, and length L. The element has two degrees of freedom at each node.

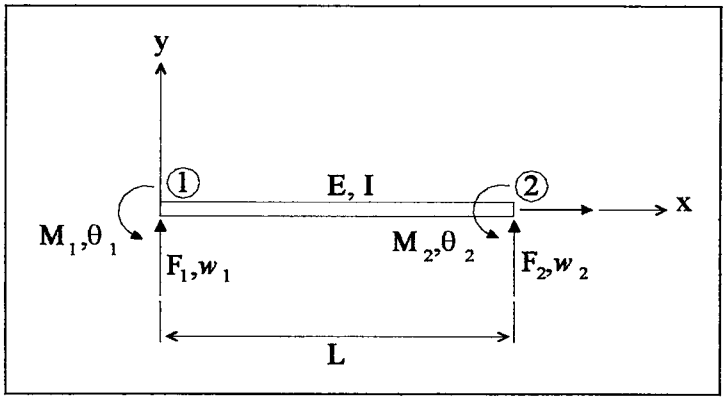


Figure 3.4 Beam Element

The well-known differential equation of equilibrium for this system is given by:

$$EI\frac{dw^4}{dx^4}=0 \tag{3.12}$$

The continuous variable w, is computed in terms of discrete nodal values. However, not only w itself, but also its derivatives are used as nodal values:

$$w(x)=N_1(x)w_1+N_2(x)\theta_1+N_3(x)w_2+N_4(x)\theta_2 \tag{3.13}$$

where

$$\theta=\frac{dw}{dx} \tag{3.14}$$

The shape functions, N_1, N_2, N_3 and N_4 are cubic Hermitian polynomials and are given by (Yang, 1986):

$$\begin{aligned}
N_1(x)&=1-3\left(\frac{x}{L}\right)^2+2\left(\frac{x}{L}\right)^3\\
N_2(x)&=x-2\left(\frac{x^2}{L}\right)+\left(\frac{x^3}{L^2}\right)\\
N_3(x)&=3\left(\frac{x}{L}\right)^2-2\left(\frac{x}{L}\right)^3\\
N_4(x)&=-\left(\frac{x^2}{L}\right)+\left(\frac{x^3}{L^2}\right)
\end{aligned} \tag{3.15}$$

Reader may refer to McGuire and Gallagher (2000) for the derivation of shape functions. The plots of shape functions are shown in Figure 3.5.

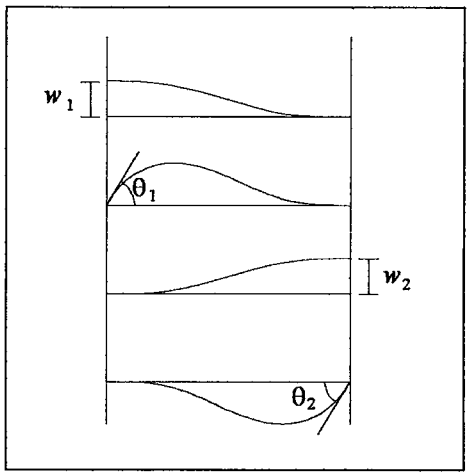


Figure 3.5 Cubic Shape Functions for the Beam Element

Using the principle of virtual work and Equation (3.13), the strain energy function for a beam element can be written as:

$$U = \frac{EI}{2}\int_0^L \left(\frac{d^2w}{dx^2}\right)^2 dx \tag{3.16a}$$

$$U = \frac{EI}{2}\int_0^L \left(\frac{d^2N_1}{dx^2}w_1 + \frac{d^2N_2}{dx^2}\theta_2 + \frac{d^2N_3}{dx^2}w_2 + \frac{d^2N_4}{dx^2}\theta_2\right)dx \tag{3.16b}$$

Performing partial differentiation of U with respect to degree of freedom w_1 gives

$$\begin{aligned} F_1 = \frac{dU}{dw_1} &= \frac{EI}{2}\int_0^L 2\left(\frac{d^2w}{dx^2}\right)\frac{d}{dx}\left(\frac{d^2w}{dx^2}\right)dx \\ &= EI\int_0^L \left(\frac{d^2N_1}{dx^2}w_1 + \frac{d^2N_2}{dx^2}\theta_2 + \frac{d^2N_3}{dx^2}w_2 + \frac{d^2N_4}{dx^2}\theta_2\right)\frac{d^2N_1}{dx^2}dx \\ &= k_{11}w_1 + k_{12}\theta_1 + k_{13}w_2 + k_{14}\theta_2 \end{aligned} \tag{3.17}$$

where:

$$k_{11} = EI\int_0^L \frac{d^2N_1}{dx^2}\frac{d^2N_1}{dx^2}dx; \quad k_{12} = EI\int_0^L \frac{d^2N_1}{dx^2}\frac{d^2N_2}{dx^2}dx$$
$$k_{13} = EI\int_0^L \frac{d^2N_1}{dx^2}\frac{d^2N_3}{dx^2}dx; \quad k_{14} = EI\int_0^L \frac{d^2N_1}{dx^2}\frac{d^2N_4}{dx^2}dx \tag{3.18}$$

Using the same procedure for other degrees of freedom we obtain the stiffness matrix as:

$$\mathbf{k} = \frac{EI}{L}\begin{bmatrix} \frac{12}{L^3} & \frac{6}{L^2} & -\frac{12}{L^3} & \frac{6}{L^2} \\ \frac{6}{L^2} & \frac{4}{L} & -\frac{6}{L^2} & \frac{2}{L} \\ -\frac{12}{L^3} & -\frac{6}{L^2} & \frac{12}{L^3} & -\frac{6}{L^2} \\ \frac{6}{L^2} & \frac{2}{L} & \frac{6}{L^2} & \frac{4}{L} \end{bmatrix} \tag{3.19}$$

These terms are the standard slope-deflection equations and the slope element stiffness matrix is the same as in equation (3.19).

In order to obtain the 6x6 stiffness matrix of frame element, truss element and beam element stiffness matrices are combined as shown in Figure 3.6.

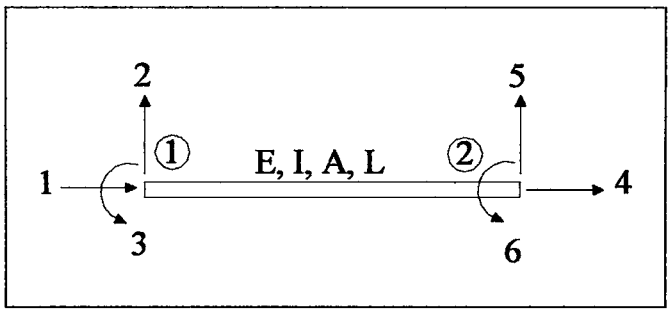


Figure 3.6 Numbering of Dofs for the Frame2D Element

Thus, the stiffness matrix becomes:

$$\mathbf{k} = \begin{bmatrix} \frac{EA}{L} & 0 & 0 & -\frac{EA}{L} & 0 & 0 \\ 0 & \frac{12EI}{L^3} & \frac{6EI}{L^2} & 0 & -\frac{12EI}{L^3} & \frac{6EI}{L^2} \\ 0 & \frac{6EI}{L^2} & \frac{4EI}{L} & 0 & -\frac{6EI}{L^2} & \frac{2EI}{L} \\ -\frac{EA}{L} & 0 & 0 & \frac{EA}{L} & 0 & 0 \\ 0 & -\frac{12EI}{L^3} & -\frac{6EI}{L^2} & 0 & \frac{12EI}{L^3} & -\frac{6EI}{L^2} \\ 0 & \frac{6EI}{L^2} & \frac{2EI}{L} & 0 & -\frac{6EI}{L^2} & \frac{4EI}{L} \end{bmatrix} \tag{3.20}$$

CALWin has also the option of forming geometric stiffness matrix. If a nonzero axial load P is applied to the beam element, the 4x4 geometric stiffness matrix of the beam is defined as

$$\mathbf{k}_g = \frac{P}{30}\begin{bmatrix} \frac{36}{L} & 3 & -\frac{36}{L} & 3 \\ 3 & 4L & -3 & -L \\ -\frac{36}{L} & -3 & \frac{36}{L} & -3 \\ 3 & -L & -3 & 4L \end{bmatrix} \tag{3.21}$$

The total stiffness of beam element is then given as:

$$\mathbf{k} = (\mathbf{k} - \mathbf{k}_g) \tag{3.22}$$

The positive sign convention of degrees of freedom and forces of plane frame element in CALWin are shown in Figure 3.7.

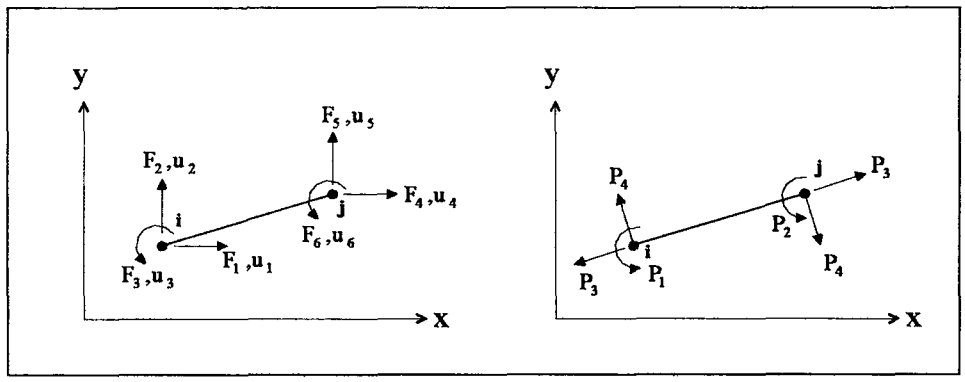


Figure 3.7 Definitions of Positive Displacements and Forces for Plane Frame Element

3.1.4 Three Dimensional Frame Element

Frame element oriented arbitrarily in three-dimensional space has 12 degrees of freedom. Such a frame element in a local coordinate system is shown in Figure 2.7.

This 12-dof-frame element can be thought as the superposition of following elements:

1. An axial force element (truss)
2. A pure torsional member
3. A beam bent about one principal axis
4. A beam bent about the other principal axis

Element stiffness matrices of beam and truss elements have already been presented. Now, we only introduce the stiffness matrix of the pure torsional member. Conceptually, the pure torsional member is identical to the axial force member. A pure torsional member is shown in Figure 3.8.

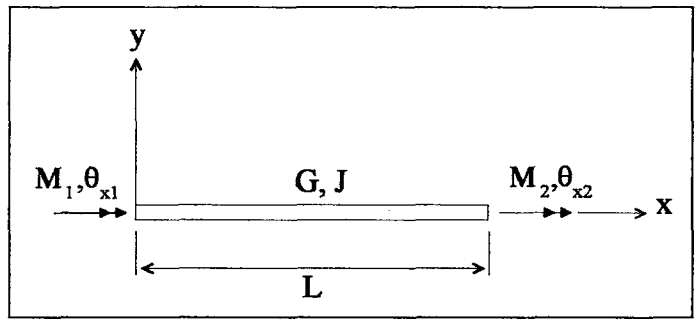


Figure 3.8 Pure Torsional Bar Element

Derivation of stiffness matrix of torsional member is also identical to that of a truss element; therefore it is not repeated here. Applying the same procedures, we obtain the stiffness matrix of torsional member as:

$$\mathbf{k} = \frac{GJ}{L}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \tag{3.23}$$

Therefore, using the stiffness matrix just derived, and the stiffness matrices of truss and beam element we 12x12 stiffness matrix can be formed by superposition.

3.1.5 Beams on Elastic Foundations

Apart from standard framework elements, CALWin also includes finite elements for beams on elastic or two-parameter elastic foundations. These elements are called BEF and BEF2P

The exact shape functions for these elements are given by Alemdar and Gülkan (1996). These shape functions are utilized to derive the analytical expressions for work equivalent nodal forces and coefficients of the stiffness, mass and geometric stiffness matrices. A very brief summary of derivation of shape functions and element matrices is presented in this section.

Before beginning, it is worth to noting that a beam segment not supported by any foundation is a special case of beam segment supported by a single parameter Winkler foundation, which in turn is a subset of the more general two-parameter case (Alemdar and Gülkan, 1996).

Free body diagram of an infinitesimal part of beam supported by a two-parameter elastic function, which terminates at the ends of the beam, is shown in Figure 3.9.

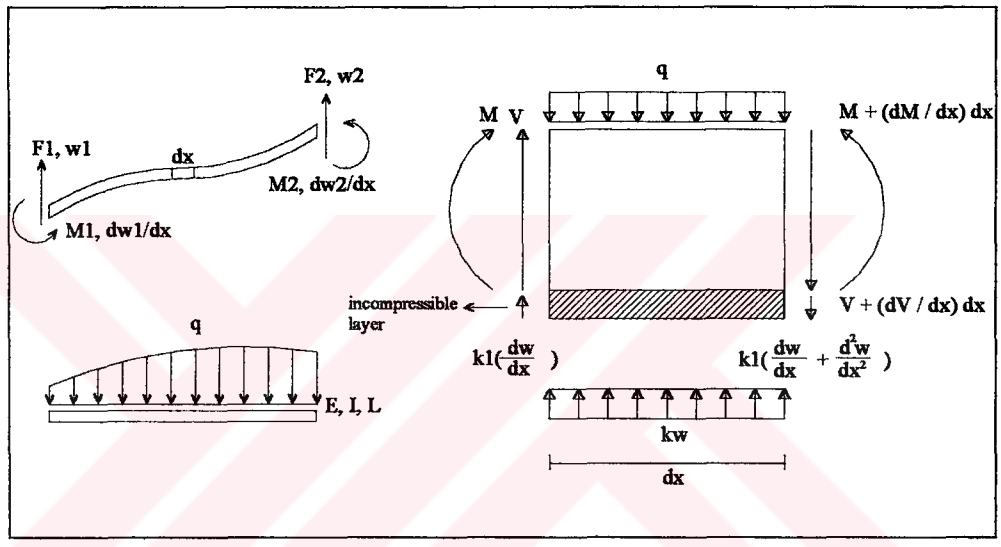


Figure 3.9 Beam Segment on a Generalized Foundation (Alemdar and Gülkan, 1996).

The parameters k and k_1 in figure are the Winkler parameter and Pasternak parameter respectively.

The governing equation may be written as

$$EI\frac{d^4w}{dx^4} - k_1\frac{d^2w}{dx^2} + kw = q(x) \tag{3.24}$$

The homogenous solution of the fourth-order differential Equation (3.24) is dependent on the relation between k_1/EI and k/EI and can be computed as:

If $k_1/EI < 2\sqrt{(k/EI)}$

$$w(x) = c_1 \cos \beta x \cosh \alpha x + c_2 \cos \beta x \sinh \alpha x + c_3 \sin \beta x \cosh \alpha x + c_4 \sin \beta x \sinh \alpha x \tag{3.25}$$

If $k_1/EI = 2\sqrt{(k/EI)}$

$$w(x) = c_1 e^{\sqrt[4]{B}x} + c_2 x e^{\sqrt[4]{B}x} + c_3 e^{-\sqrt[4]{B}x} + c_4 x e^{-\sqrt[4]{B}x} \tag{3.26}$$

If $k_1/EI > 2\sqrt{(k/EI)}$

$$w(x) = c_1 \cosh \beta x \cosh \beta \alpha x + c_2 \sinh \beta x \cosh \alpha x + c_3 \cosh \beta x \sinh \alpha x + c_4 \sinh \beta x \sinh \alpha x \tag{3.27}$$

where

$$\alpha = \sqrt{\lambda^2 + \delta} \tag{3.28}$$

$$\beta = \sqrt{\lambda^2 - \delta} \text{ in Equation (3.25)} \tag{3.29}$$

$$\beta = \sqrt{\delta - \lambda^2} \text{ in Equation (3.27)} \tag{3.30}$$

$$\lambda = \sqrt[4]{\frac{k}{4EI}} \text{ and } \delta = \frac{k_1}{4EI} \tag{3.31}$$

If $k_1=0$ all three expressions for $w(x)$ in equations(3.25) to (3.27) reduce to

$$w(x) = c_1 \sin \lambda x \sinh \lambda x + c_2 \sin \lambda x \cosh \lambda x + c_3 \cos \lambda x \sinh \lambda x + c_4 \cos \lambda x \cosh \lambda x \tag{3.32}$$

Equations 3.21 may be written in matrix form:

$$w = \mathbf{NW} \tag{3.33}$$

where

$$\mathbf{N} = \mathbf{B}^{\mathrm{T}}\mathbf{H}^{-1} \quad \text{and} \quad \mathbf{W} = \mathbf{HC} \tag{3.34}$$

Here **N** is the row vector containing the four shape functions. **C** is the column vector containing the integration constants. **W** is the column vector containing the four boundary conditions and **H** is a matrix of coefficients to be determined.

The expressions for shape functions N_i, i = 1 - 4 are very lengthy and cumbersome. For this reason they are not presented in this text. However, detailed expressions are given in Alemdar (1995).

Once the shape functions are evaluated, the element matrices are obtained with the standard procedure. The 4x4 element stiffness matrix **k** is given by:

$$k_e = EI\int \frac{d^2\mathbf{N}^T}{dx^2}\frac{d^2\mathbf{N}}{dx^2}dx + k_1\int \frac{d\mathbf{N}^T}{dx}\frac{d\mathbf{N}}{dx}dx + k\int \mathbf{N}^T\mathbf{N}dx \qquad (3.35)$$

The 4x4 consistent mass matrix **m** for dynamic analysis involving a uniform segment with mass per unit length μ is given by:

$$m = \mu\int \mathbf{N}^T\mathbf{N}dx \qquad (3.36)$$

Consistent geometric stiffness matrix $\mathbf{k_g}$ for a constant compressive axial force P is given by:

$$k_g = P\int \frac{d\mathbf{N}^T}{dx}\frac{d\mathbf{N}}{dx}dx \qquad (3.37)$$

3.2 Two Dimensional Elements

So, far formulations of only one-dimensional finite elements were presented. In this section, formulation of two-dimensional elements plane stress-strain elements that were added to finite element library of CAL in the scope of this study.

First, a summary of isoparametric finite element formulation is given. Then, the governing equations of plane stress and plane strain problems are stated.

3.2.1 Isoparametric Formulation of Finite Elements

Isoparametric element formulation was introduced in 1968 by Irons (1968). It allowed very accurate, higher-order elements of arbitrary shape to be developed and programmed with minimum effort (Wilson, 1998).

In the isoparametric finite element formulation the element coordinates and element displacements are expressed in the form of interpolations using the natural coordinate system of the element (Bathe, 1996). For the two-dimensional case, natural coordinates are denoted by r and s, each vary from –1 to +1. Considering the two dimensional case coordinate interpolations can be written as:

$$x = \sum_{i=1}^{q} h_i x_i; \qquad y = \sum_{i=1}^{q} h_i y_i \tag{3.38}$$

where x_i and y_i are the coordinates of the q element nodes and h_i are the interpolation functions of variables r and s.

The name "isoparametric" implies that the element displacements are interpolated in the same way as the geometry. For the two dimensional case the displacements in the direction of element local axes, u and v, is given by:

$$u = \sum_{i=1}^{q} h_i u_i; \qquad \mathrm{v} = \sum_{i=1}^{q} h_i v_i \tag{3.39}$$

The fundamental property of interpolation function h_i is that its value in the natural coordinate system is unity at node i, and zero at all other nodes. In Figure 3.10 4 to 9 variable-number-nodes two-dimensional curved element is shown (Bathe 1996).

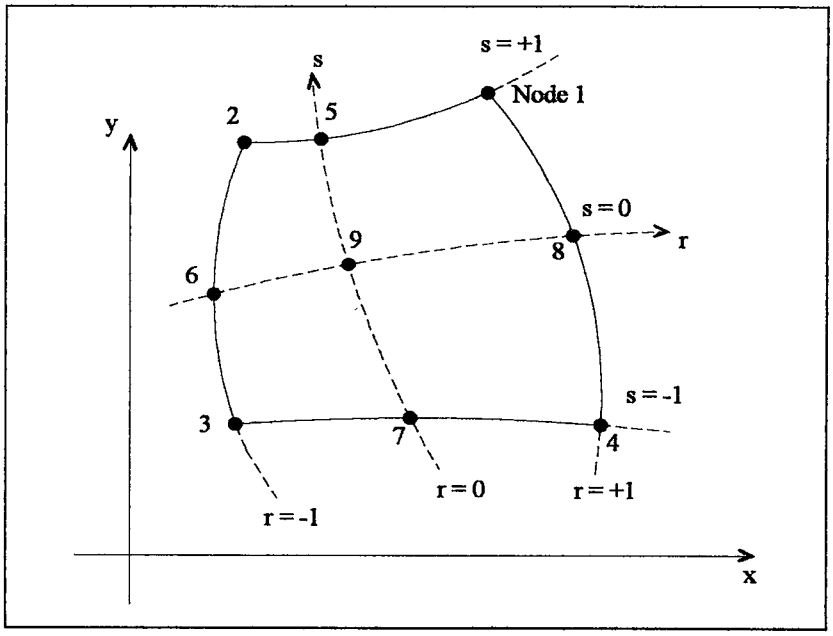


Figure 3.10 A 4 to 9 Variable-Number-Nodes Two-Dimensional Element (Bathe, 1996)

The hierarchic shape functions for the 4 to 9 node two-dimensional element is given in Table 3.1 (Bathe, 1996). If any node from 5 to 9 of the element shown in Figure 3.10 is missing, the shape functions associated with that node are zero and are not calculated. Note that the sum of all shape functions is always equal to 1.0 for all points within the element (Wilson, 1998).

Table 3.1 Interpolation Functions of 4 to 9 Variable-Number-Nodes Two-Dimensional Element (Bathe, 1996).

		Include only if node i is defined				
		$i=5$	$i=6$	$i=7$	$i=8$	$i=9$
H_1 =	$\frac{1}{4}(1+r)(1+s)$	$-\frac{1}{2}h_5$			$-\frac{1}{2}h_8$	$-\frac{1}{4}h_9$
H_2 =	$\frac{1}{4}(1-r)(1+s)$	$-\frac{1}{2}h_5$	$-\frac{1}{2}h_6$			$-\frac{1}{4}h_9$
H_3 =	$\frac{1}{4}(1-r)(1-s)$		$-\frac{1}{2}h_6$	$-\frac{1}{2}h_7$		$-\frac{1}{4}h_9$
H_4 =	$\frac{1}{4}(1+r)(1-s)$			$-\frac{1}{2}h_7$	$-\frac{1}{2}h_8$	$-\frac{1}{4}h_9$
H_5 =	$\frac{1}{2}(1-r^2)(1+s)$					$-\frac{1}{2}h_9$
H_6 =	$\frac{1}{2}(1-s^2)(1-r)$					$-\frac{1}{2}h_9$
H_7 =	$\frac{1}{2}(1-r^2)(1-s)$					$-\frac{1}{2}h_9$
h_8 =	$\frac{1}{2}(1-s^2)(1+r)$					$-\frac{1}{2}h_9$
h_9 =	$(1-r^2)(1-s^2)$					

In order to calculate the stiffness matrix of an element, the strain-displacement matrix need to be calculated. The element strains are obtained in terms of derivatives of displacements with respect to x and y. Because the element displacements are defined in the natural coordinate system, x, y derivatives of displacement are related to r, s derivatives using the chain rule:

$$\begin{aligned}\frac{\partial u}{\partial r} &= \frac{\partial u}{\partial x}\frac{\partial x}{\partial r}+\frac{\partial u}{\partial y}\frac{\partial y}{\partial r}\\ \frac{\partial u}{\partial s} &= \frac{\partial u}{\partial x}\frac{\partial x}{\partial s}+\frac{\partial u}{\partial y}\frac{\partial y}{\partial s}\end{aligned} \quad \text{or} \quad \begin{bmatrix}\frac{\partial u}{\partial r}\\ \frac{\partial u}{\partial s}\end{bmatrix} = \mathbf{J}\begin{bmatrix}\frac{\partial u}{\partial x}\\ \frac{\partial u}{\partial y}\end{bmatrix} \tag{3.40}$$

Where **J** is the Jacobian matrix given by:

$$\mathbf{J} = \begin{bmatrix} \frac{\partial x}{\partial r} & \frac{\partial y}{\partial r} \\ \frac{\partial x}{\partial s} & \frac{\partial y}{\partial s} \end{bmatrix} \tag{3.41}$$

Then, the derivatives of displacements with respect to x and y can be evaluated using:

$$\begin{bmatrix} \frac{\partial u}{\partial x} \\ \frac{\partial u}{\partial y} \end{bmatrix} = \mathbf{J}^{-1} \begin{bmatrix} \frac{\partial u}{\partial r} \\ \frac{\partial u}{\partial s} \end{bmatrix} \tag{3.42}$$

This requires that the inverse of **J** exists. The inverse does exists provided that there is a one-to-one correspondence between the natural and the local coordinates of the element. However, if the element is much distorted or folds back upon itself, the unique relation between coordinate systems does not exist, and the matrix **J** is singular. Provided that the inverse exists, **J** can be inverted numerically at the integration points.

Using stress-strain relations and equations (3.38) to (3.42) the element stiffness matrix can be calculated from:

$$\mathbf{K} = t\int_{S} \mathbf{B}^T \mathbf{C}\mathbf{B} dS \tag{3.43}$$

Where **B** is the strain-displacement transformation matrix and **C** is the material matrix, and t is the thickness of the element. In Equation (3.43), the surface integral dS should be written in terms of natural coordinates. So, we have:

$$dS = \det \mathbf{J} \cdot dr \cdot ds \tag{3.44}$$

3.2.2 Plane Stress and Plane Strain Elements

In plane stress problems a two-dimensional stress situation exists in the xy plane. As a result, τ_{zz}, τ_{yz} τ_{zx} are equal to zero. Plane stress elements are used for modeling membranes, the in-plane action of beams and walls and thin plates. Plain strain elements, on the other hand, are used to represent a slice of a structure in which the strain components ε_{zz}, γ_{yz} and γ_{zx} are zero. This situation arises, e.g., in the analysis of a long dam (Bathe, 1996).

The governing equations of the two types of plane elasticity problems differ from each other only in the constitutive (stress-strain) relations (Reddy, 1993).

Equations of motion for plane stress and plane strain problems including the dynamic case are:

$$\begin{aligned} \frac{\partial \sigma_x}{\partial x} + \frac{\partial \tau_{xy}}{\partial y} + f_x &= \rho \frac{\partial^2 u}{\partial t^2} \\ \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \sigma_y}{\partial y} + f_x &= \rho \frac{\partial^2 v}{\partial t^2} \end{aligned} \tag{3.45}$$

where f_x and f_y are the body forces per unit volume and ρ is the density of the material.

Strain displacement relations are given by:

$$\varepsilon_x = \frac{\partial u}{\partial x}; \quad \varepsilon_y = \frac{\partial v}{\partial y}; \quad \gamma_{xy} = \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \tag{3.46}$$

Stress-strain (or constitutive) relations:

$$\begin{bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{bmatrix} = \mathbf{C} \begin{bmatrix} \varepsilon_x \\ \varepsilon_y \\ \gamma_{xy} \end{bmatrix} \tag{3.47}$$

where **C** is the material matrix and given as for the plane stress condition:

$$\frac{E}{1-v^2}\begin{bmatrix} 1 & v & 0 \\ v & 1 & 0 \\ 0 & 0 & \frac{1-v}{2} \end{bmatrix} \tag{3.48}$$

The material matrix of plane strain element is

$$\frac{E(1-v)}{(1-v)(1+2v)}\begin{bmatrix} 1 & \frac{v}{1-v} & 0 \\ \frac{v}{1-v} & 1 & 0 \\ 0 & 0 & \frac{1-2v}{2(1-v)} \end{bmatrix} \tag{3.49}$$

where E is the modulus of elasticity and v is the Poisson's ratio.

CHAPTER 4

DYNAMIC ANALYSIS PROCEDURES

In this chapter the theoretical background of procedures that are available in CALWin, for solving problems related to dynamic analysis of structures are given.

Along with other commands in CALWin, it is possible to solve the following types of dynamic problems:

- Evaluation of free vibration mode shapes and frequencies.
- Automatic generation of Ritz vectors to be used in a mode superposition analysis or response spectra analysis.
- Mode superposition analysis due to arbitrary loading
- Step-by-step analysis of structural systems with arbitrary viscous damping
- Dynamic analysis in the frequency domain.

4.1 Numerical Evaluation of Mode Shapes and Frequencies

The first step in the solution of dynamic response of the structures is the determination of the natural frequencies and mode shapes. The natural frequencies of ω_n and modes φ_n must satisfy the following algebraic equation:

$$\mathbf{k}\phi_n = \omega_{n^2}\mathbf{m}\phi_n \tag{4.1}$$

This equation is known as the generalized eigenvalue problem in which the eigenvalues are $\lambda_n \equiv \omega_n^2$

The eigenvalues λ_n, are the roots of characteristic equation:

$$P(\lambda) = \det[\mathbf{k} - \lambda\mathbf{m}] = 0 \tag{4.2}$$

Where P(λ) is a polynomial of order N, the number of degrees of freedom of the system. This is not a practical method especially for large systems. We need more efficient and reliable methods for the solution of eigenvalue problem (Chopra, 2001).

Finding reliable and efficient methods to solve the eigenvalue problem has been the subject of much research and many methods have been developed (Bathe, 1996)

These methods can be classified into three major categories depending on the nature of the solution algorithm:

- Vector iteration methods
- Transformation methods
- Polynomial iteration techniques

Combinations of two or more methods have been developed also to handle the very large systems. Lanczos method and subspace-iteration method are the example of such combined procedures (Bathe, 1996)

The eigen solution method implemented in CALWin is the JACOBI method, which is one of the transformation methods. Jacobi method transforms the matrix $[\mathbf{k} - \lambda\mathbf{m}]$ into a diagonal matrix with successive rotations.

The Jacobi method is not efficient for systems with size larger than 100 degrees of freedom. Since CALWin is designed for solution of small educational and research problems this is not a major problem. The Jacobi method implemented in CALWin is a modified Jacobi method where both **K** and **M** must be symmetric and positive definite.

4.2 Dynamic Response By Mode Superposition

The second phase in evaluating the dynamic response of a structural system, after determining the natural frequencies and mode shape, is the direst mode superposition analysis.

In mode superposition analysis the coupled equation of motion are transformed to modal coordinates, leading to an uncoupled set of modal equations. Each modal equation is solved to determine the modal contributions to the response, and these modal responses combined to obtain the total response. The procedure is as follows:

The equations of motion for an N-DOF system with damping are given by:

$$\mathbf{m}\ddot{\mathbf{u}} + \mathbf{c}\dot{\mathbf{u}} + \mathbf{k}\mathbf{u} = \mathbf{p}(t) \tag{4.3}$$

Replacing the displacement with modal coordinates

$$\mathbf{u}(t) = \sum_{i=1}^{N} \phi_i q_i(t) = \mathbf{\Phi}\mathbf{q}(t) \tag{4.4}$$

Substituting Equation (4.4) into Equation (4.3) gives

$$\sum_{i=1}^{N} \mathbf{m}\boldsymbol{\phi}_i \ddot{q}_i(t) + \sum_{i=1}^{N} \mathbf{c}\boldsymbol{\phi}_i \dot{q}_i(t) + \sum_{i=1}^{N} \mathbf{k}\boldsymbol{\phi}_i q_i(t) = \mathbf{p}(t) \tag{4.5}$$

Premultiplying each term with ϕ_j^T gives

$$\sum_{i=1}^{N} \boldsymbol{\phi}_j^T \mathbf{m}\boldsymbol{\phi}_i \ddot{q}_i(t) + \sum_{i=1}^{N} \boldsymbol{\phi}_j^T \mathbf{c}\boldsymbol{\phi}_i \dot{q}_i(t) + \sum_{i=1}^{N} \boldsymbol{\phi}_j^T \mathbf{k}\boldsymbol{\phi}_i q_i(t) = \boldsymbol{\phi}_j^T \mathbf{p}(t) \tag{4.6}$$

Because of the orthogonality property of mode shapes, acceleration and velocity terms in each of the summations vanish except when i = j reducing equation (4.6) to

$$M_i \ddot{q}_i(t) + \sum_{j=1}^{N} c_{ij} \dot{q}_i(t) + K_i q_i(t) = p_i(t) \tag{4.7}$$

where $$M_i = \boldsymbol{\phi}_i^T \mathbf{m}\boldsymbol{\phi}_i \tag{4.8}$$

$$K_i = \boldsymbol{\phi}_i^T \mathbf{k}\boldsymbol{\phi}_i \tag{4.9}$$

$$P_i = \boldsymbol{\phi}_i^T \mathbf{p}\boldsymbol{\phi}_i \tag{4.10}$$

and $$C_{ji} = \boldsymbol{\phi}_i^T \mathbf{c}\boldsymbol{\phi}_j$$

Equation (4.7) can be written for N equations in matrix form:

$$\mathbf{M}\ddot{\mathbf{q}} + \mathbf{C}\dot{\mathbf{q}} + \mathbf{K}\mathbf{q} = \mathbf{P}(t) \tag{4.11}$$

Where **M**, **K** and **P**(t) are diagonal and **C** is a non-diagonal matrix. The N equations will be uncoupled if the system has classical damping for which

$$C_{ij} = 0 \text{ for } I \neq j \tag{4.12}$$

Therefore equation (4.7) reduces to

$$M_i \ddot{q}_i + C_i \dot{q}_i + K q_i = P_i(t) \tag{4.13}$$

The solution of N uncoupled dynamic equations gives modal coordinates. Once the modal coordinates have been determined, total displacements are evaluated using Equation (4.4).

4.3 Step-by-Step Direct Integration

Analytical solution of the uncoupled modal equations of motion of a multi degree of freedom system is usually is not possible if the excitation varies arbitrarily with time or the system is nonlinear (Chopra, 2001). Furthermore, uncoupling of modal equations is not possible if the system has non-classical damping (Chopra, 2001). For such systems the dynamic response is obtained through step-by-step integration methods (time stepping methods).

The Newmark method is one o f the most flexible step-by-step integration methods (Wilson, 1991). This method is based on the following equations:

$$\dot{u}_{i+1} = \dot{u}_i + [(1-\gamma)\Delta t]\ddot{u}_i + (\gamma\Delta t)\ddot{u}_{i+1} \tag{4.14}$$

$$u_{i+1} = u_i + (\Delta t)\dot{u}_i + [(0.5-\beta)\Delta t^2]\ddot{u}_i + [\beta(\Delta t)^2]\ddot{u}_{i+1} \tag{4.15}$$

Where β and γ are constants selected to define the variation of acceleration over a time step and to determine the desired stability and accuracy. If γ= ½ and β=1/6, linear acceleration is produced. If γ= ½ and β= ¼ constant average acceleration is produced. However, this is an unconditionally stable method without numerical damping (Wilson, 1991)

The solution algorithm is presented in Chopra, 2001.

Another method is the Wilson's Method developed by Wilson (1991). The Wilson θ-method is a technique which can be used to modify the basic Newmark method in order to increase stability limits and to add numerical dumping. In the θ

method if the θ=1, the method reverts to the linear acceleration method. If θ≥1,37 this method becomes unconditionally stable.

In CALWin, Newmark-Wilson method has been integrated. The values of parameters γ, β and θ that control the acceleration and damping are summarized in Table 4.1.

Table 4.1 Parameters for the Newmark-Wilson Method

	γ	β	θ
Newmark's Average Acceleration	1/2	¼	1.00
Newmark's Linear Acceleration	1/2	1/6	1.00
Theta Method – Low Damping	1/2	1/6	1.42
Theta Method – High Damping	1/2	1/6	2.00

4.4 Frequency-domain Analysis

The frequency-domain method is an alternative method to the time domain method. In frequency-domain the time function is eliminated from the equations using the Fourier transform method.

The first step in frequency-domain analysis is to express the excitation function in frequency space:

$$P(w) = F[p(t)] = \int_{-\infty}^{\infty} p(t) e^{-iwt} dt \qquad (4.16)$$

The Fourier transform U(ω) of the solution U(t) of the differential equation is then given by

$$U(w) = H(w) P(w) \qquad (4.17)$$

Where H(ω) is the complex frequency response function. H(ω) describes the response of the system to harmonic excitation. Finally the described solution U(t) is given by the inverse Fourier transform of U(ω):

$$u(t)=\frac{1}{2\pi}\int_{-\infty}^{\infty} H(w)P(w)e^{iwt}dw \qquad (4.18)$$

The direct and inverse Fourier transforms must be evaluated numerically by the discrete Fourier transform method using fast Fourier transform algorithm.

The frequency-domain method is especially useful and powerful for dynamic analysis of structures interacting with unbounded media. Another advantage of the frequency-domain approach is in its application to the substructure analysis (Wilson, 1991).

CHAPTER 5

IMPLEMENTATION OF CALWIN

5.1. Introduction

In the preceding chapters, the underlying theory of the analysis procedure implemented in CALWin was briefly explained assuming that the reader is familiar with the basic language. The aim of succeeding work is to explain the implementation of CALWin.

As stated previously, the primary focus of this study is to develop a user-friendly analysis program built on top of the capabilities of CAL91. For this reason, a short review of the development tools that were used for the development of CALWin is given first. This discussion includes a short review of object oriented programming with C++ language, the OpenGL graphics library and the principles of Windows programming with MFC. Then, an overview of CALWin software is presented including the new features that were added. Next, general structure and object model of the program is presented. Finally, the class hierarchies of the CALWin are explained briefly.

5.2 Development Tools

5.2.1 C++ Language and Object Oriented Programming

5.2.1.1 What Is C++ and Why Is It The Choice?

Stroustrup (1997) stated that "a programming language serves two related purposes: it provides a vehicle for the programmer to specify actions to be executed, and it provides a set of concepts for the programmer to use when thinking about what can be done. The first purpose ideally requires a language that is "close to machine" so that all important aspects of a machine are handled simply and efficiently in a way that is reasonably obvious to the programmer. The second purpose ideally requires a language that is "close to the problem to be solved" so that the concepts of a solution can be expressed directly and concisely."

Therefore, C++ can be defined as a general-purpose programming language with a bias towards systems programming that

- is a better C,
- supports data abstraction,
- supports object-oriented programming, and
- supports generic programming.

Apart from the advantages stated above, C++ provides much more flexibility. First of all it is the most widely used programming language in the world, it can be compiled on almost every operating system and it has very powerful tools. Secondly, C++ has all the advantages of Fortran with the help of Standard Template Library and many more features that Fortran does not provide.

Finally, it is important to make the programs long lasting in the rapidly changing environment of computers. In other words, a program should be readable, understandable and even compilable for many years.

In conclusion, C++ was chosen as the programming language for the development of CALWin instead of Fortran or Visual Basic due to the stated reasons.

5.2.1.2 Object Oriented Programming Paradigm

All programming languages provide abstractions. Assembly language is a small abstraction of the underlying machine. Many so-called "imperative" languages that followed (such as Fortran, BASIC, and C) were abstractions of assembly language. These languages are big improvements over assembly language, but their primary abstraction still requires one to think in terms of the structure of the computer rather than the structure of the problem that he or she is trying to solve. The programmer must establish the association between the machine model and the model of the problem that is actually being solved. The effort required to perform this mapping, and the fact that it is extrinsic to the programming language, produces programs that are difficult to write and expensive to maintain. The alternative to modeling the machine is to model the problem to be solved (Eckel B., 2000).

Early languages such as LISP and APL chose particular views of the world ("All problems are ultimately lists" or "All problems are algorithmic" (Eckel B., 2000)). PROLOG casts all problems into chains of decisions. Languages have been created for constraint-based programming and for programming exclusively by manipulating graphical symbols. (The latter proved to be too restrictive.) Each

of these approaches is a good solution to the particular class of problem they're designed to solve, but when you step outside of that domain they become awkward.

The object-oriented approach provides tools for the programmer to represent elements in the problem space. This representation is general enough that the programmer is not constrained to any particular type of problem. Elements in the problem space and their representations in the solution space are referred as "objects". The idea is that the program is allowed to adapt itself to the nature of the problem by adding new types of objects, so when one reads the code describing the solution, her or she is reading words that also express the problem. This is a more flexible and powerful language abstraction. Thus, OOP (Object Oriented Programming) allows one to describe the problem in terms of the problem, rather than in terms of the computer where the solution will run. There's still a connection back to the computer, though. Each object looks quite a bit like a little computer; it has a state, and it has operations that you can ask it to perform.

There are five basic characteristics of object orientation (Page-Jones, 2000):

- Encapsulation
- Information/implementation hiding
- Classes
- Inheritance
- Polymorphism

According to Page and Jones:

Encapsulation is the grouping of related ideas into one unit, which can thereafter be referred to by a single name. Object-oriented encapsulation is the packaging of operations and attributes representing state into an object type so that state is accessible or modifiable only via the interface provided by the encapsulation.

Information/implementation hiding is the use of encapsulation to restrict from external visibility certain information or implementation decisions that are internal to the encapsulation structure.

A class is a template from which objects are created (instantiated). Objects instantiated from the same class share the same structure and behavior.

Classes may form inheritance hierarchies of base classes and child classes. Inheritance allows objects of one class to use the facilities of any of the base classes of that class. Also, operations of a class may be redefined (overridden) in a child class.

Polymorphism is the facility by which a single operation name may be defined upon many different classes and may take on different implementations in each of those classes.

All of these characteristics of OOP were extensively used in the development of CALWin. Especially inheritance and polymorphism are vital characteristics of finite element and visual element class hierarchies and MFC (Microsoft Foundation Classes) itself.

5.2.2 OpenGL

5.2.2.1 What Is OpenGL?

OpenGL is a library of subroutines that form an interface to graphics hardware. This library consists of about 250 distinct commands that can be used to specify the objects and operations needed to produce interactive three-dimensional applications.

OpenGL is designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms. To achieve these qualities, no commands for performing windowing tasks or obtaining user input are included in OpenGL; instead, one must work through whatever windowing system controls the particular hardware her or she is using. Similarly, OpenGL doesn't provide high-level commands for describing models of three-dimensional objects. Such commands might allow one to specify relatively complicated shapes such as buildings, parts of the body, airplanes, or molecules. With OpenGL, you must build up your desired model from a small set of geometric primitives – points, lines and polygons.

5.2.2.2 OpenGL Rendering Pipeline

Most important characteristic of the OpenGL is that it produces three-dimensional graphics incredibly quickly. The main reason is that OpenGL does not eliminate hidden lines and surfaces using complicated intersection and collision calculations, instead a frame buffer is used in which the color of each pixel on the screen is stored.

OpenGL performs the operations in a series of processing stages called the OpenGL rendering pipeline. This ordering, as shown in Figure 5.1, is not a strict

rule of how OpenGL is implemented but provides a reliable guide for predicting what OpenGL will do (Woo, 1999).

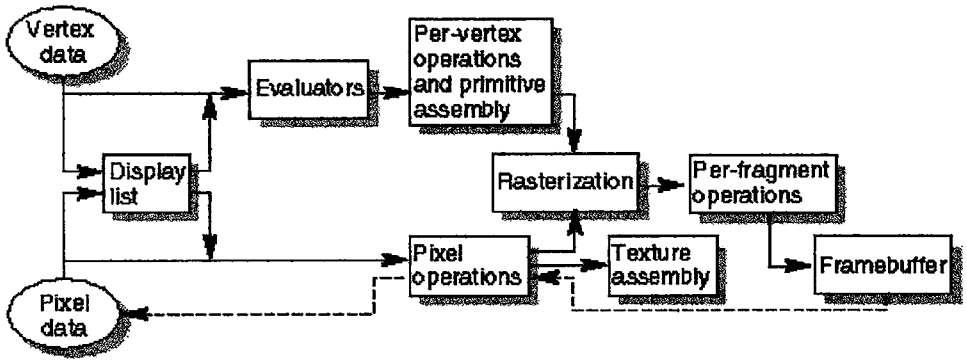


Figure 5.1 OpenGL Rendering Pipeline (Woo, 1999)

Some of these stages were not processes in rendering of CALWin. Because there are no textures, display lists or parameterized curves or surfaces in CALWin. Therefore all of these stages will not be explained here. Only the parts of this figure, which are important parts of the rendering of CALWin, were explained.

Firstly, vertex data is read into OpenGL, next is the "Per-vertex operations" stage, which converts the vertices into primitives. Some types of vertex data (for example, spatial coordinates) are transformed by 4x4 floating-point matrices. Spatial coordinates are projected from a position in the 3D world to a position on to screen.

Next stage is the primitive assembly. Clipping, a major part of primitive assembly, is the elimination of portions of geometry that fall outside of the visible area of the model. In some cases, this is followed by perspective division, which makes distant geometric objects appear smaller than closer objects. Then viewport and depth (z-coordinate) operations are applied. The results of this stage are

complete geometric primitives, which are the transformed and clipped vertices with related color, depth for the rasterization step.

Rasterization stage is the conversion of both geometric and pixel data into fragments. Fragments are squares corresponding to pixels in the frame buffer.

Final stage is the "per-fragment" operations. Depth test performed on this stage. If a fragment cannot pass depth test it is not stored in frame buffer. Therefore, at the end of this we have stage only the visible fragments, and these fragments are stored in frame buffer.

5.2.2.3 State Management and Drawing

Another important characteristic of OpenGL is that, it is a state machine. This means that OpenGL is put it into various states (or modes) that remain in effect until it is changed. For example, the current color can be set to white, red, or any other color, and thereafter every object is drawn with that color until the current color is set to something else. The current color is only one of many state variables that OpenGL maintains. For example, current viewing and projection transformations, line and polygon stipple patterns, polygon drawing modes are all state variables. This state management saves OpenGL from repeating unnecessary operations, increasing the speed of rendering.

5.2.2.4 Viewing and Modeling Transformations

Viewing and modeling transformations are the most important part of rendering a three-dimensional model in OpenGL. OpenGL has three main transformations; modeling transformation, projection transformation, and viewport transformation. However, OpenGL maintains two transformation matrices in the background; modelview matrix and Projection matrix.

Modeling transformation is used to position and orient the model. For example, one can rotate, translate, or scale the model or perform some combination of these operations.

Projection transformation is used for three-dimensional world coordinates to two-dimensional screen coordinates. Projection transformation determines the viewing volume and to some extent the look of the objects. There are two projection types in OpenGL. One is the perspective projection and the other is orthographic transformation.

Viewport transformation together with the projection transformation determines how a scene is mapped onto the screen. The projection transformation specifies the mechanics of how the mapping should occur, and the viewport indicates the shape of the available screen area into which the scene is mapped.

More information on the subject can be found in Woo, 1999.

5.2.3 MFC and Windows Programming

5.2.3.1 General Concepts of Windows Programs

A Windows program is quite different from a typical DOS program, and it is more complicated.

Firstly, a DOS program has the control of the hardware and DOS operating system is subservient. When a service is requested by the program (for example an input operation), an operating system function is called. It is even possible to bypass the operating system and access the PC hardware directly. A Windows program, however, is subservient and Windows is in control of the hardware. Windows program can only access the hardware by way of Windows

functions. No direct access to these hardware resources is permitted, because several programs can be active at one time under Windows. Windows has to determine which application is the destination of a given input, and inform that program accordingly.

Secondly, in a Windows application a range of different inputs are possible at any given time. A user may press a key, select a menu item or click the mouse somewhere in the application window. A well-designed Windows application must be prepared to deal with any type of input at any time. The user actions are regarded as events and will result in a particular piece of program code to be executed. How program execution proceeds is therefore determined by the sequence of user actions. Programs that operate in this way are called event-driven programs.

Therefore, a windows program consists of pieces of code that respond to events caused by the actions of user (Horton, 1998). Structure of such a program is shown in the figure below:

As a result of handling all possible types of events, even an elementary Windows program involves many lines of code. CALWin as a "small-size" Windows application consists of more than 40000 lines of code.

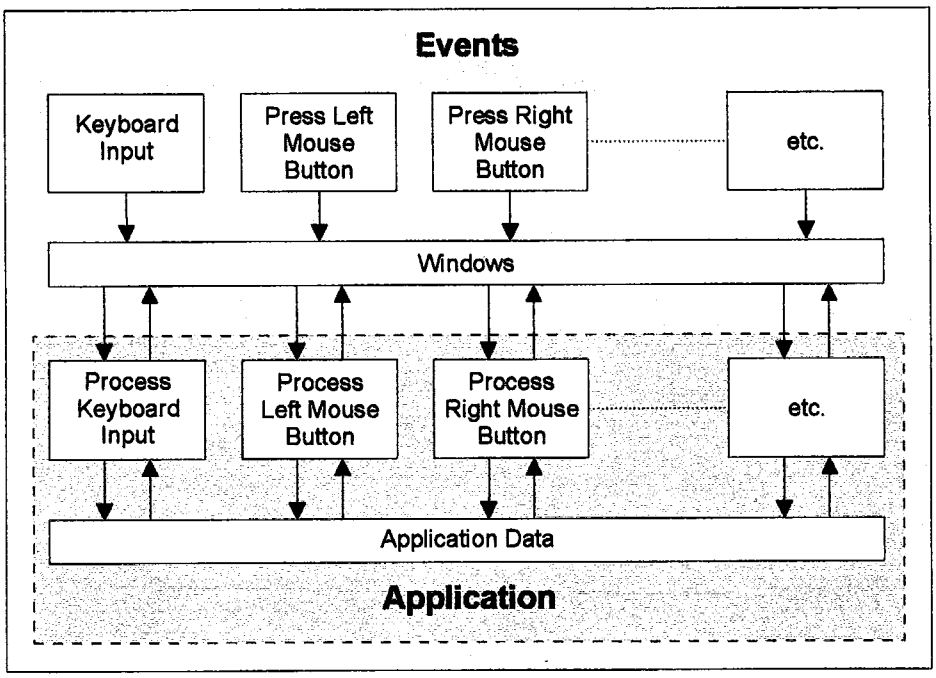


Figure 5.2 Structure of a Typical Windows Program (Horton, 1998)

5.2.3.2 What is MFC

MFC stands for Microsoft Foundation Classes. The Microsoft Foundation Classes are a set of C++ classes. These classes represent an object-oriented approach to Windows Programming that encapsulates the Windows API.

A Windows program based on MFC creates and uses MFC objects and/or objects of classes derived from MFC. The objects created will have member functions for communicating with Windows, for processing Windows messages and for sending messages to other objects. For more information on the subject see Blaszczak, 1999.

5.2.3.3 Document / View Architecture

An application's code can be divided into two; the code that manages application's data and the code that presents this data to the user. Both of these parts of code are dependent on each other (Blaszczak, 1999).

In the document/view model, the class that manages the data is called "document". Although the word "document" implies something of textual, it is not limited to text. In fact, it could be anything the programmer wants. The document encapsulates the data the application manipulates. In MFC, document is represented by a document class.

The code that represents the data to the user is called the view. MFC represents a view with a view class. The view is also responsible for interacting with user. Since the view is the window that the user perform actions with the mouse and keyboard. A document may have more than one view, each presenting the different part of the document.

MFC incorporates a mechanism for integrating a document with its views. A document object maintains a list of pointers to its associated views and a view object has a data member holding a pointer to the document.

5.2.3.4 Serialization

Serialization means writing application data to storage or reading it from storage. Serialization is embedded into the base object of MFC. Therefore, all MFC objects derived from CObject have the serialization ability. To serialize an application specific object, it must be derived from CObject class, directly or indirectly. Then it must override Serialize() function of the CObject class.

5.3. Overview of the CALWin Software

CALWin is a Windows based structural analysis software. The "CAL" in the name stands for "Computer Assisted Learning" and "Win" stands for "Windows based".

CALWin has been designed as an educational tool. Unlike many other automated structural analysis software, CALWin provides an interface to the user for performing manually the individual steps of analysis procedure of structural systems. At the same time, it gives opportunities, which are not available in its predecessor CAL91, such as a powerful graphical interface and ability to automate selected steps.

When an analysis is performed or a problem is solved in CALWin, the solution procedure can be divided into three phases as shown in Figure 5.3.

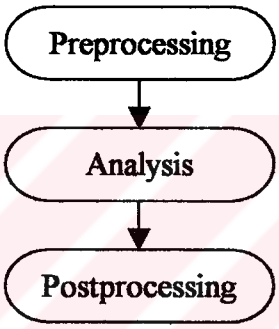


Figure 5.3 Basic Steps of Analysis.

The preprocessing phase includes the definition of the geometry of structure, definition of the degrees of freedom, definition of the loads. Analysis phase can be thought as composed of computation of element matrices, assembly of global equations and solution of these equations. Determination of member forces, plot of time variation of displacements, viewing of member force diagrams and deflected shape of the structure constitute the post-processing phase.

All of these operations in each of these phases can be performed using dialog boxes in CALWin. This is the manual usage of the CALWin. Alternatively, some of these operations can be automated using the graphic editor. For example, if the structure and loads are defined using the graphic editor, the computation of

element and structural matrices, and solution of equations are carried out automatically. The running of the program and the dialog boxes are explained in the next chapter.

A general view of CALWin is shown in Figure 5.4.

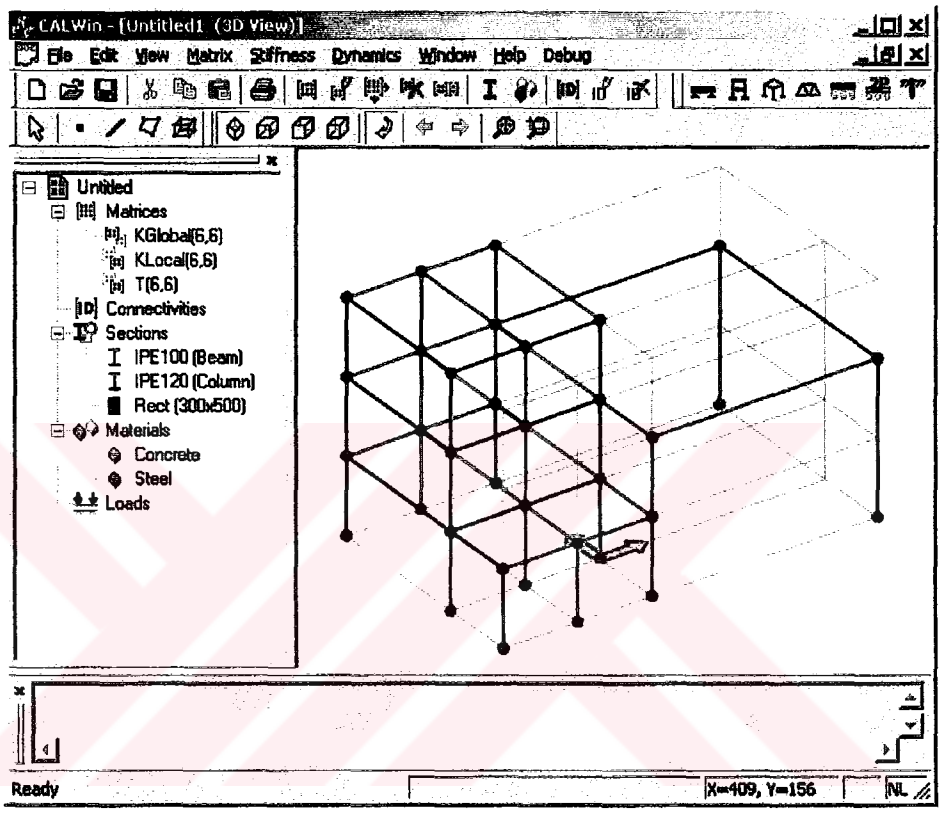


Figure 5.4 General View of CALWin

CALWin application window has three main components, apart from the standard Windows components, which are the menu bar and the toolbars. These three main components are:

- The output window
- The tree view
- Graphic editor windows

The output window is the part at the bottom of the window in Figure 5.4. It displays the error messages and warnings as well as progress messages.

On the left in Figure 5.4, is the tree view. Tree view is an important component of the user interface that facilitates the data input and management. All matrices, material properties, section properties and loads defined in the document are displayed in and accessible through the tree view. Of course it is possible to perform any job without using the tree view.

Tree view and output window are dockable controls and can be docked to any side of the application's main window.

The graphic editor is the heart of the term "user-friendliness" that was mentioned in the introduction chapter. Graphic editor windows are used for defining the structure with the mouse. With the help of OpenGL supported graphics engine, it is possible to view the structure and to define the elements in any two- or three-dimensional viewing angle. There can be more than one view open at any time in the application.

CALWin is an MDI (Multiple Document Interface) application, which means that it is possible to work with more than one document at the same time in CALWin.

The analysis commands in CALWin is separated into three main categories. These are matrix operations, direct stiffness operations and dynamics operations.

Since the matrix structural analysis is based on matrices, matrix operations are the basic operations for all analysis problems. Matrix operations implemented in CALWin are given in Table 5.1.

Table 5.1 Matrix Operations in CALWin.

Matrix Operations	Matrix Operations
Multiplication	Matrix Operation Dialog
Transpose multiplication	Matrix Operation Dialog
Addition	Matrix Operation Dialog
Subtraction	Matrix Operation Dialog
Transpose	Matrix Operation Dialog and matrix editor
Duplication	Matrix Operation Dialog
Storing on diagonal of a matrix	Matrix Operation Dialog
Duplication of diagonal of a matrix	Matrix Operation Dialog
Scaling with a scalar	Matrix Operation Dialog
Inversion	Matrix Operation Dialog and matrix editor
Solution of system of equations	Matrix Operation Dialog
Storing a sub-matrix	Matrix Operation Dialog
Duplication of a sub-matrix	Matrix Operation Dialog

Direct stiffness operations are used for calculating element stiffness matrices (depending on the element type, this may include geometric stiffness and mass matrices), assembling these matrices to the system matrices and evaluating member forces. These operations are listed in Table 5.2.

Dynamics operations include the various dynamic analysis procedures, which are presented in Table 5.3.

Table 5.2 Direct Stiffness Operations in CALWin

Direct Stiffness Operations	Accessible From
Slope (Beam) element matrices	Menu and toolbar
2D Frame element matrices	Menu and toolbar
Beam on Elastic Foundation element matrices	Menu and toolbar
Beam on 2P Elastic Foundation element matrices	Menu and toolbar
Pile element matrices	Menu and toolbar
Truss element matrices	Menu and toolbar
3D Frame element matrices	Menu and toolbar
Triangle Plane Stress/Strain element matrices	Menu and toolbar
Quadrilateral Plane Stress/Strain element matrices	Menu and toolbar
Definition of Connectivity Array	Menu and toolbar
Assembly of element matrices	Menu
Evaluation of member forces	Menu

Table 5.3 Structural Dynamics Operations in CALWin

Dynamic Operations	Accessible From
Eigenvalue solution for diagonal mass matrix	Menu
Eigenvalue solution with Jacobi method	Menu
Square root of matrix terms	Matrix Operation Dialog
Inverse of matrix terms	Matrix Operation Dialog
Uncoupled dynamic response	Menu
Combination (ABS, SRSS)	Matrix Operation Dialog
Generation of time function	Menu
Maximum ABS	Matrix Operation Dialog
Stepwise numerical integration	Menu
Plot of response	Menu and Toolbar
Calculation of Ritz vectors	Menu
Discrete Fourier Transform	Menu
Inverse DFT	Menu
Radius	Menu
Frequency Domain Solution	Menu

5. 4. General Structure and Object Model of CALWin

In this chapter, basic components that form the structure of the computer program are explained. Firstly, the structure of the program is divided into groups and these are explained. Then the relations between objects are explained with the help of a diagram. Finally, those exceptions to object oriented design of the CALWin are stated.

CALWin has been developed using C++ with an object-oriented approach. The structure of the program consists of about 140 classes that contain approximately 40,000 lines of C++ code. In a broad sense, these classes can be categorized into three groups;

- Classes that implement the structural statics and dynamics and all related procedures; *Computational Classes.*
- Classes that provides a visual or non-visual interface to analysis objects, which can be just matrices or structural members such as columns, beams; *Interface Classes.*
- Classes that make the connection between first two types of classes and Windows operating system; *Dialog and Application Framework Classes.*

5.4.1. Computational Classes

Classes in the first group form the basis of all the computation carried out in CALWin. All data of the analysis is stored in these classes and all operations performed on analysis data is defined in these classes. These classes can be subdivided into following groups.

- Basic Linear Algebra classes
- Finite Element Classes
- Solver Classes

5.4.1.1 Basic Linear Algebra classes

In CAL91, all static and dynamic analysis procedures are carried out by means of matrices and matrix operations. One of the main objectives while developing CALWin was to preserve this idea. For that reason, a set of linear algebra classes has been designed. These were the first classes developed and they were used extensively in the design of all other classes as data members or function arguments.

There are four classes in the category, but the most important of these is the CMatrix class. This class represents the abstraction of an NxM matrix where N>0 and M>0. Class implements all matrix operations including the inversion of a square matrix. Basic operations like matrix addition or matrix multiplication are implemented through operator overloading allowing expressions similar to the ones in textbooks. Detailed explanation of the CMatrix class and other classes are given in section 5.5.2.

5.4.1.2 Finite Element Classes

Finite element classes are the abstraction of structural members such as columns, beams on generalized foundations or plane elements in plane stress or strain conditions. Finite element classes are derived from a common abstract base class. This allowed the creation of a flexible class hierarchy upon which new finite elements can easily be added.

An important point about these classes is that these classes do not implement or manage the visual appearance and behavior of the structural members they represent, nor do they contain the data (for example the coordinates of the end nodes) required to define those elements. These classes only define the direct stiffness operations of elements and are used to compute element matrices (mass, stiffness or geometric stiffness matrix).

5.4.1.3 Solver Classes

This group includes a number of solvers for linear system of equations for different characteristics, like symmetric banded system of equations or general case of unsymmetrical system of equations.

5.4.2. Interface Classes

These classes define interfaces to analysis objects, which were explained in the previous section. The objects seen in the graphic editor or matrices displayed in matrix editor are instances of these classes. Beside these visible objects, classes that do not represent visual objects themselves but being used indirectly in the visualization of these objects are also included in this category. This category can further be subdivided into three:

- Visual Element Classes
- Analysis Element Classes
- Smart Matrix Classes
- OpenGL Rendering Classes

5.4.2.1 Visual Element Classes

These classes are used to define the structural model. Nodes, line elements (beams, columns, etc.) and surface elements (for example shear walls) are all instances of these classes. Visual element classes have the ability to interact with the user, and are responsible from the drawing of the elements they represent. Because of this reason, these classes contain code that makes call to OpenGL routines.

5.4.2.2 Smart Matrix Classes:

Smart matrix classes are not matrices actually. They are used as a container for the actual matrix class CMatrix. However, these containers serve to an important purpose; awareness and adaptivity. Awareness means that an object of smart matrix class is aware of other objects of this class if they have been linked to, and adaptivity means that these objects automatically recalculate themselves if one of the objects on which they depend on changes. This is similar to the case of cells in an Excel spreadsheet linked through a formula. If value of one of the cells changes, formula is calculated automatically and result cell is updated. This process is done via a messaging system between objects. Each object maintains a list of objects on which it depends and a list of objects that depend on it.

There are two types of smart matrix classes. First one is the CSmartMatrix class. Second type is the matrix operation classes and contains a different class for each matrix operation.

5.4.2.3 Analysis Elements Classes

These classes were developed to represent some properties used in the analysis model. These include the loads, structural materials, sections for the line elements and element connectivity matrices. In other words, these classes store the data of the analysis model. Some of them have a visual appearance in the graphic editor.

5.4.2.4 OpenGL Rendering Classes

These classes are responsible from the rendering of the model on to display. They do some necessary operations that required for OpenGL functions to run and initialize the scene setting appropriate values for model space and projection parameters. Some of these classes are responsible from object and point selection operations mapping two-dimensional screen coordinates to three-dimensional model coordinates using projection transformations.

5.4.3 Dialog and Application Framework Classes

As discussed in the previous section, a Windows program consists primarily of pieces of code that respond to events caused by the actions of the user. The classes in this category process these events. They can be divided into two sub-groups: Dialog classes and Application Framework classes. All classes in both categories are derived from MFC (Microsoft Foundation Classes) classes. Dialog classes are mainly used for collecting data from the user that is specific to a selected action. Application framework classes on the other hand initialize the application, start the message-processing loop and process these messages. Another responsibility of application framework classes is to manage the data of

the application (in this case analysis model data) and assure the persistence of this data.

5.4.4 Object Relations

Figure 5.5 illustrates the relationships between objects in the program. It is important to note that, this diagram is neither a flow chart nor an inheritance tree, and it only includes important objects. The solid arrows represent links from an object to other. Dashed arrows represent a derivation from an object.

As seen from the diagram, all objects are in some way connected to document object. Document object is responsible from the connection between objects. Application object accepts messages dispatched by the Windows and redirects them to the appropriate objects; main window object, dialog objects, document object and child window objects. Besides these relations, analysis model objects maintain links to objects that are related themselves instead of accessing them using the document object. Another direct relation is that, each scene object maintains a relation list to the analysis model objects that are visible in that window.

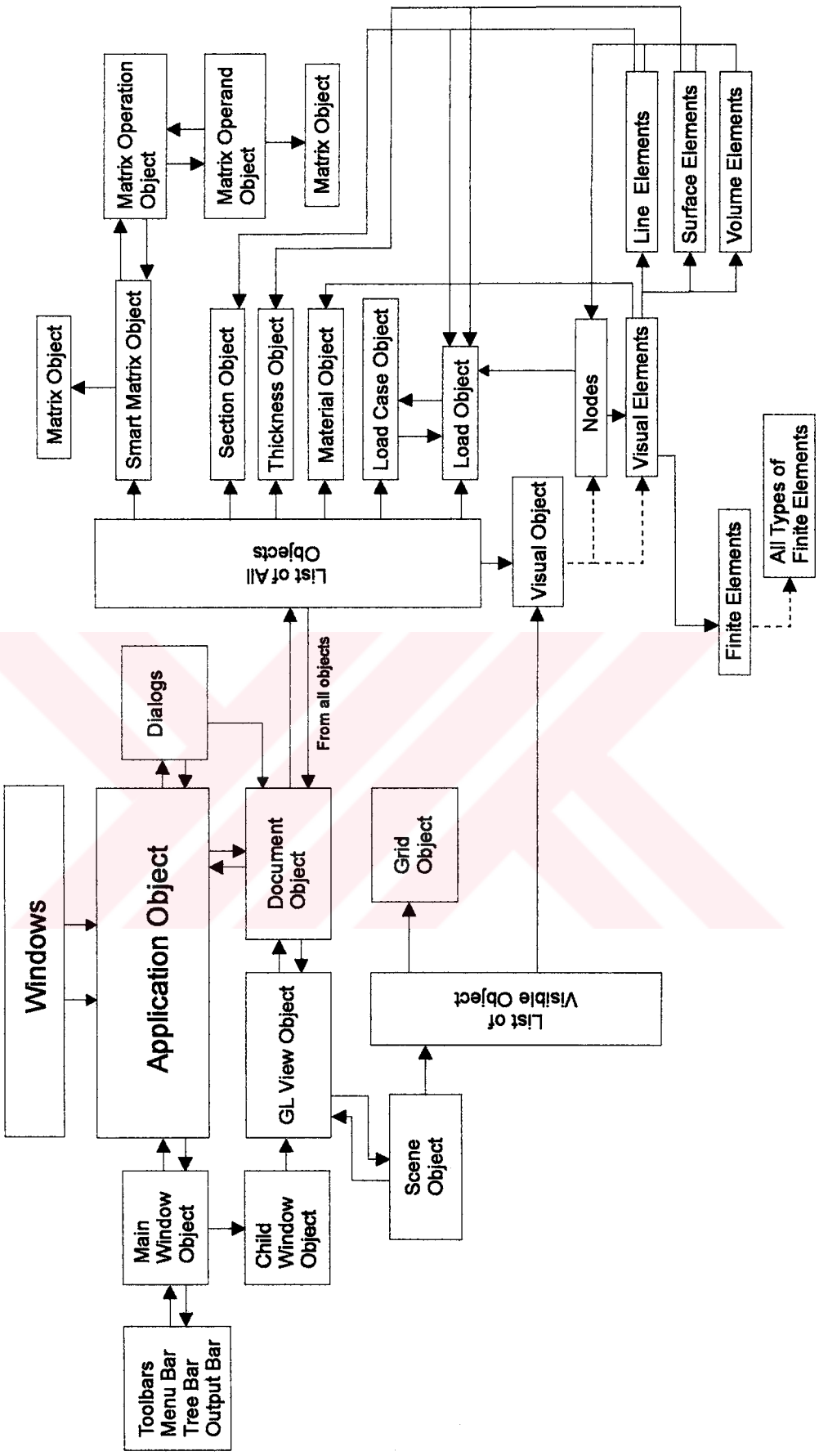


Figure 5.5 Object Relations

5.4.5 Deviations from Object Oriented Programming Approach

Although an object-oriented approach was utilized in general in the development process of CALWin, there are some areas in which this approach was avoided. These are the analysis routines imported from CAL91. A list of these routines is given in Table 5.4:

Table 5.4 CAL91 Subroutines Converted to C++ Code

CalEigen	Solves eigenvalue problem with diagonal mass matrix
CalJacobi	Solves eigenvalue problem with full mass matrix
CalFunct	Generates a time function at equal intervals
CalDynam	Performs a modal analysis
CalStep	Stepwise numerical integration
CalGauss	Gauss elimination routine for symmetric matrices
CalRitz	Computes Ritz vectors
CalDFT	Discrete Fourier Transform and Inverse DFT
CalFSolve	Solution in frequency domain

Since these routines consist of an algorithm only and do not have a data to manage, an object-oriented approach wouldn't be much useful in this case. For this reason they are coded as functions rather than classes.

An important point here is that, the algorithms are modified in such a way that all types of GOTO statements are eliminated from the algorithms using different type of control structures and loops. The motivation for this is to eliminate a bad-programming style from the routines. In modern programming paradigm, GOTO statements are considered as a bad-programming style since they transfer the execution from one point to another and severely decreases code maintainability and readability.

5.5 Class Hierarchies

A class itself as a programming artifact does not have much meaning. Because objects are not isolated from each other rather they interact with each other. Hence, the design of a class depends on other classes. Classes that are related or dependent to each other forms class hierarchies.

In this section class hierarchies of the CALWin are explained. Inheritance diagrams of these classes are also given in this section.

CALWin consists of over 120 classes as stated previously. These classes are grouped into following categories:

- Application Framework Classes
- Basic Linear Algebra Classes
- Base Class
- Visual Element Classes
- Finite Element Classes
- Analysis Element Classes
- Solver Classes
- OpenGL Rendering Classes
- Dialog Classes
- Control Classes
- Third Party Control Classes

Now, important classes in these categories are explained one by one. However, before presenting the detailed explanations, I would like the make a note about the naming convention. Although there is no rule or necessity imposed by the C++ compiler on the class names, names of all classes in CALWin start

with capital "C" by convention. This convention is a matter of personal taste and a result of my programming habits.

5.5.1 Application Framework Classes

As the name implies, classes in this category forms the framework of the program. The application framework is responsible for helping the application to initialize, for running it by providing it with the capability of responding messages and terminating the application.

These classes are common to all windows applications developed using MFC (Microsoft Foundation Classes) with document-view architecture. These classes were derived from MFC classes and customized to suit the requirements of CALWin. Classes in this category are:

- CPhantomApp
- CMainFrm
- CChildFrm
- CDoc
- CGLView

5.5.1.1 CPhantomApp Class

This class defines the core application object. Processes messages and command line parameters. An object of this class is constructed before the windows of the application are created.

CPhantomApp is derived from MFC class CWinApp as shown in the inheritance diagram.

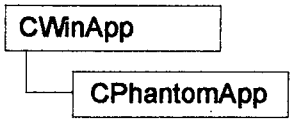


Figure 5.6 Inheritance Diagram for CPhantomApp Class

5.5.1.2 CMainFrame Class

This class defines the main window of the application. Derived from MFC class CMDIFrameWnd, which provides the functionality of a Windows multiple document interface (MDI) frame window, along with members managing the window.

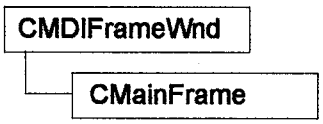


Figure 5.7 Inheritance Diagram for CMainFrame Class

5.5.1.3 CChildFrame Class

This class defines the Windows multiple document interface (MDI) child window. Derived from CMDIChildWnd class, which provides the functionality of a MDI child window, along with members for managing the window. An MDI child window looks much like a typical frame window, except that the MDI child window appears inside an MDI frame window rather than on the desktop. An MDI child window does not have a menu bar of its own, but instead shares the menu of the MDI frame window.

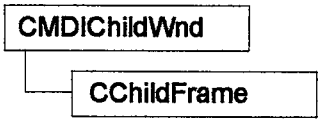


Figure 5.8 Inheritance Diagram for CChildFrame Class

5.5.1.4 CDoc Class

This class defines the document object. It is one of the most important classes in the class hierarchy of CALWin because it was developed using document/view architecture of the MFC and document lies in the heart of this design.

The document class manages the data manipulated by the application. The data is rendred for the user by the view class, which will be explained in the next section. Document class also provides serialization support. Serialization means saving the data to a file on the disk or loading from a file on the disk.

CDoc class is derived from the MFC class CDocument as shown in the inheritance diagram.

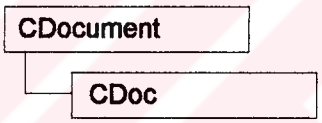


Figure 5.9 Inheritance Diagram for CDoc Class

CDoc class stores the application data in list of objects. The protected data members define the STL vectors (lists) of different types of objects. These lists are populated by the constructors and destructors of objects that will be added to corresponding lists. In other words, an object adds itself to the appropriate list of CDoc class when it is constructed and removes itself from the same list when it is destroyed. Objects manipulate these lists through the public member functions of the CDoc class. For example AddObject() and RemoveObject() methods.

Another important method of the CDoc class is the Serialize() function. This function is called by the application framework to save or load the document.

CDoc class also provides some type definitions that can be used by other classes. The important ones of these are DocState and ModelSpace types. DocState type is the state of the document at any given time. This is used by the objects to determine whether document is being destroyed or not. If the document is being destroyed, objects do not try to remove themselves from the lists to which they were added.

5.5.1.5 CGLView Class

CGLView class renders the structural model data. CGLView class is derived from the MFC class CView. CView class is attached to the document and acts as an intermediary between the document and the user.

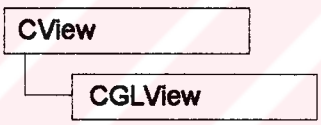


Figure 5.10 Inheritance Diagram for CGLView Class

This class is also responsible for interacting with the user since the view is the window that the user manipulates directly with the mouse and keyboard.

Most important data member of the CGLView class is the scene object which is an instance of CScene class. The OpenGL drawing is accomplished by this object. CGLView class acts as a container for the scene object and supplies the data required by the scene object to render the model.

Zoom, real time rotate and real time pan operations, selection operations, element creation operations and deletion operations are carried out by the CGLView class.

As a result of performing all these operations, this class is a lengthy class (approximately 3000 lines of code in implementation file).

5.5.2 Basic Linear Algebra Classes

These classes implement matrices, vectors and their operations. These classes were utilized throughout the whole program whenever a matrices and vectors were needed. The analysis routines specified Section 5.4.5 operates on objects of these classes. The interface of these classes and overloaded operators were inspired from the utility classes of another object oriented structural analysis software called OpenSEES (web site: www.opensees.ce.berkeley.edu). However, implementation details of these classes are different then those of OpenSEES.

There are four classes in this category:

- CMatrix
- CVector
- CVertex3
- CID

5.5.2.1 CMatrix Class

This class is one of the core classes of CALWin. It defines an NxM matrix where N and M are integers greater than zero. CMatrix is a stand-alone class. It does not have a parent or child classes.

All matrix-matrix, matrix-vector and matrix-scalar operations are implemented using operator overloading. These operators are shown in Figure 5.10.

```
.
.
//Fortran-like index operators
inline double& operator()(int row, int col);
inline double  operator()(int row, int col) const;

//Matrix-scalar operators (operate on self)
CMatrix& operator+=(double fact);
CMatrix& operator-=(double fact);
CMatrix& operator*=(double fact);
CMatrix& operator/=(double fact);

//Matrix-scalar operators (create new matrix)
CMatrix  operator+(double fact) const;
CMatrix  operator-(double fact) const;
CMatrix  operator*(double fact) const;
CMatrix  operator/(double fact) const;

//Matrix - vector operations
CVector operator*(const CVector &V) const;
CVector operator^(const CVector &V) const;

//Matrix-Matrix operations (create new matrix)
CMatrix  operator+(const CMatrix& other) const;
CMatrix  operator-(const CMatrix& other) const;
CMatrix  operator*(const CMatrix& other) const;
CMatrix  operator^(const CMatrix& other) const;

//Matrix-Matrix operations (operate on self)
CMatrix& operator=(const CMatrix& other);
CMatrix& operator=(const CVector& aVector);
CMatrix& operator+=(const CMatrix& other);
CMatrix& operator-=(const CMatrix& other);

//Unary Matrix operators
CMatrix  operator-() const;
```

Figure 5.11 Overloaded Operators for CMatrix Class

The use of these operators allows very short matrix expressions to be used in the code, which are very readable and similar to the ones in textbooks. To illustrate this, suppose that we have three matrices with names **K**, **A** and **k** where **K** is defined as:

$$\mathbf{K} = \mathbf{A}^T\mathbf{k}\mathbf{A} \tag{5.1}$$

Then, the C++ code would be something like this

```
K = A^k*A;
```

Where "^" is the transpose multiplication operator and "*" is the normal multiplication operator. Semi colon is required for the syntax of C++.

Other operations on matrices, like inversion, transposing, interchanging rows and columns are implement through member functions.

5.5.2.2 CVector Class

This class implements a vector of dimension N. CVector class is a stand-alone class. It does not have a parent or child classes. The interface of this class is similar to that of CMatrix class. All vector operations are implemented using operator overloading.

5.5.2.3 CVertex3 Class

CVertex3 class is used for defining points in 3-dimensional space. However, the usage of class is not limited coordinate storage. It is also used as a three-dimensional vector. It defines addition, subtraction, cross product and dot product of three-dimensional vectors using operator overloading. Objects of class are utilized in coordinate system calculations.

5.5.2.4 CID Class

This class implements an integer array. Objects of this class are used for storing and passing equation numbers of nodes.

5.5.3 Base Class of Analysis Model Objects: CBase

Every class in CALWin that is a part of the analysis model is derived from CBase class. CBase class has two important roles in CALWin. First, it implements a couple of important functions that are common to all classes and

links all objects of its derived classes to the document object. Second, it provides access to all analysis domain objects without knowing the exact type of it. This mechanism is called polymorphism.

CBase class is derived from the MFC's CObject class in order to implement serialization mechanism. The inheritance diagram of CBase class is shown in Figure 5.11. Figure 5.12 displays interface of CBase class.

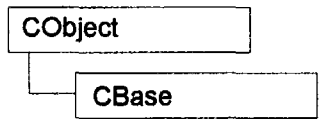


Figure 5.12 Inheritance Diagram for CBase Class

```
class CBase : public CObject
{
public:
    //Type definitions
    enum ObjectState {
        osNormal,
        osDelete,
    };

protected:
    DECLARE_SERIAL(CBase)          //Create from serialization
    CBase() {}                     //Default constructor is not
accessible
public:
    CBase(CDoc* theDoc);           //Constructor
    virtual ~CBase();              //Destructor

    //Attributes
    UID         GetUniqueID() const {return m_uniqueID;}
    CDoc*       GetDocument() {return m_pDoc;}
    ObjectState GetObjectState() const {return
m_objectState;}
    virtual void Serialize(CArchive& ar);

protected:
    //Class data
    UID         m_uniqueID;
    CDoc*       m_pDoc;
    ObjectState m_objectState;
};
```

Figure 5.13 Interface of CBase Class

CBase class provides following services to its derived classes:

- Holds a pointer to document object. This pointer is used for accessing to the document by the object or other objects and functions that operate on it.
- Obtains a unique ID for the object. This parameter is used to re-construct the pointer relations between objects when the document is loaded from a file.
- Registers object to the document on creation. Adds object automatically to the object list maintained in the document. This list is used by the document object to serialize its data.
- Detaches object from document on deletion. When an object is deleted, automatically removes it from the object list of the document.
- Changes objects state to deletion upon a call to destructor. Some objects implements a messaging mechanism between each other. Sometimes it is necessary to bypass this mechanism. Object state is used for ignoring messages.
- Serialization support. Serializes the object.

5.5.4. Visual Element Classes

These classes define the objects that the user creates in the graphic editor. These classes are derived from a common base class, which is the CVisualObject class, and they must override a member function to render themselves.

There are seven classes in this category. These are:

- CVisualObject
- CGrid
- CNode
- CVisualElement
- CLineElement
- CSurfaceElement
- CVolumeElement

5.5.4.1 CVisualObject Class

Principle base class of all visual objects. Derived from CBase. The inheritance diagram of CVisualObject class is shown in Figure 5.13.

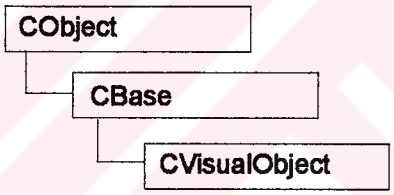


Figure 5.14 Inheritance Diagram for CVisualObject Class

Because of its pure virtual functions, CVisualObject is an abstract base class, which means that no object of this class can be instantiated. Derived classes must implement those pure virtual functions. Most important of these functions is the OnDisplaySelf(). This function is called by the application every time the display is updated. Derived classes override this function to draw themselves into the frame buffer of OpenGL.

5.5.4.2 CGrid Class

CGrid class defines a grid system similar to that of SAP2000. Grid is used as a visual aid and starting point to define the structure in the graphic editor. In fact, it is impossible to snap to points in a 3D view without the grid if there is no node defined at that point.

CGrid class is derived from CVisualObject class and overrides its pure virtual functions. The inheritance diagram of CGrid class is shown in Figure 5.14

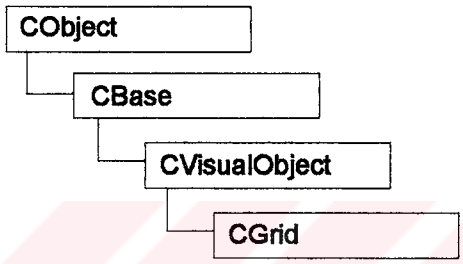


Figure 5.15 Inheritance Diagram for CGrid Class

Every document has one and only one grid object. It is created automatically when the document is created and deleted when the document is destroyed. Therefore, no command is supplied to the user for creating and deleting grids. However, user can set the number of grid lines and their positions.

5.5.4.3 CNode Class

This class defines the nodal points of the elements. It is derived from CVisualObject class. Inheritance diagram of CNode class is shown in Figure 5.15.

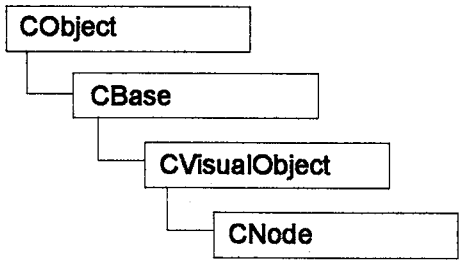


Figure 5.16 Inheritance Diagram for CNode Class

Nodes are created by the line, surface and volume elements automatically. However, it is also possible to create nodes manually. Automatically created nodes are deleted automatically when all of the connected elements are deleted. This is accomplished by a messaging mechanism between visual elements and nodes.

CNode class has a user defined unique ID other than the application defined unique ID.

5.5.4.4 CVisualElement Class

This is an abstract base class for line, surface and volume elements. This class provides a common interface to these three classes. This common interface consists of a set of pure virtual functions for querying characteristics of associated finite element object.

CVisualElement class is derived from CVisualObject class as shown in Figure 5.16.

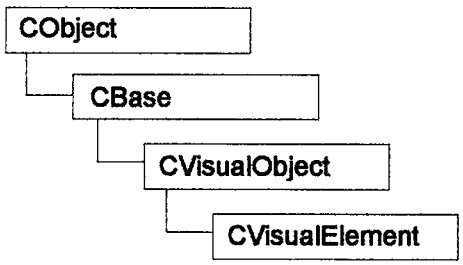


Figure 5.17 Inheritance Diagram for CVisualElement Class

5.5.4.5 CLineElement Class

This class is the abstraction of general framework element. It is derived from CVisualElement class as shown in Figure 5.17

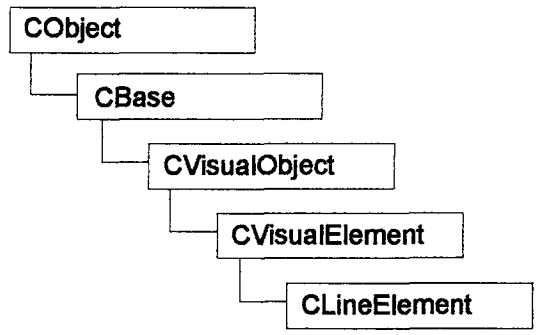


Figure 5.18 Inheritance Diagram for CLineElement Class

This class does not implement functionality of a finite element. Instead it holds a link to a finite element object. In this way it is possible to change the type of finite element without deleting element and creating an element of the appropriate type. Such an approach would cause side effects. For example, if there were load objects associated with it, they would also be deleted resulting in a loss of model data. Finite element of CLineElement class can be set to any two-noded element.

5.5.4.6 CSurfaceElement Class

This class defines the two-dimensional structural members. Actually surface elements are three-dimensional elements but one of the dimensions, which is the thickness, is small compared to other sides. As for the CLineElement class, CSurfaceElement class is also derived from CVisualElement class.

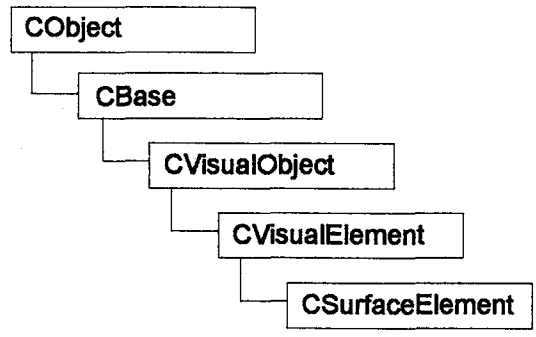


Figure 5.19 Inheritance Diagram for CSurfaceElement Class

Surface element object stores a link to a finite element object. This finite element object can be a CQuad or CTriad class object at the moment. However, new two-dimensional elements, for example a plate element, can be coded and used with the surface element. Number of nodes of the surface element depends on the number of nodes of the associated finite element.

5.5.4.7 CVolumeElement Class

This class has not been implement in CALWin. However, it is put into the hierarchy and can be used for further studies.

5.5.5 Finite Element Classes

These are the classes that define real finite elements that are used to calculate element matrices. In CALWin, finite elements are always dependent onto appropriate CVisualElement-derived class. All finite element classes are derived from a common base class, which is the CFiniteElement class. This class provides a common interface for all finite elements.

CFiniteElement class does not have a base class. Since it is used as a run-time class, it does not have to persistent and therefore is not derived from CBase class.

5.5.5.1 CFiniteElement Class

Base class of all finite element classes. It does not derived from any other class. It is an abstract class. Thus, no object of this class can be instantiated.

This class contains all the functions required to get an important data of a finite element. The derived classes must implement all of these functions so that it returns its own properties and element matrices.

5.5.5.2 Other Finite Element Classes

There are nine finite element classes in CALWin. All of them are derived from CFiniteElement class and have the same interface with CFiniteElement class. They will not be explained separately. These finite element classes are CBeam, CBEF, CBEF2P, CFrame2D, CFrame3D, CPile, CTruss, CQuad, CTriad.

5.5.6 Analysis Element Classes

These classes define remaining analysis elements other than visual elements, for example materials, frame sections, surface thickness properties, etc. These elements do not have visual appearances. Therefore, they are not derived from CVisualObject class. Although loads have visual appearances they are included in this category because visual objects to which they are applied render them. Most important of these classes are CAssembler, CConnectivityMatrix, CCoorSys and CAnalysisIntegrator classes.

5.5.6.1 CAssembler Class

CAssembler class is used for assembling structural stiffness matrix, mass matrix and connectivity matrix. This class is a stand-alone class. It is used as a temporary object. In other words, it is created just to calculate the matrices mentioned above, and is deleted just after it finishes its task.

5.5.6.2 CConnectivityMatrix Class

This class defines the connectivity matrix for the element matrix assembly. This class is derived from the CBase class as shown in Figure 5.19. Hence, it is capable of serializing itself.

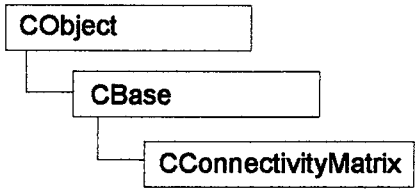


Figure 5.20 Inheritance Diagram for CConnectivityMatrix Class

Number of rows of connectivity matrix represents the number of elements. Number of columns represents the number of degrees of freedom per element.

5.5.6.3 CCoorSys Class

This class is used for defining a right-handed Cartesian coordinate system. It is a stand-alone class. This class provides member functions for creating a coordinate system from three points or two vectors, for rotating and coordinate system about its axes, and for transforming given points to global or local coordinates. It is also able to render itself although it is not derived from CVisualObject class.

This class is used for CVisualObject class, its derived classes and CFiniteElement-derived classes.

5.5.6.4 CAnalysisIntegrator Class

This class performs all automated analysis jobs including the static and eigenvalue analyses and handles the requests coming from CNode class objects for obtaining analysis results.

5.5.7 Solver Classes

A set of equation solver classes was implemented in CALWin. This set consists of four linear equation solvers and an eigen solver. However, these

classes were derived from a couple of base classes to provide a common interface. These classes with the inheritance relations are shown in Figure 5.20.

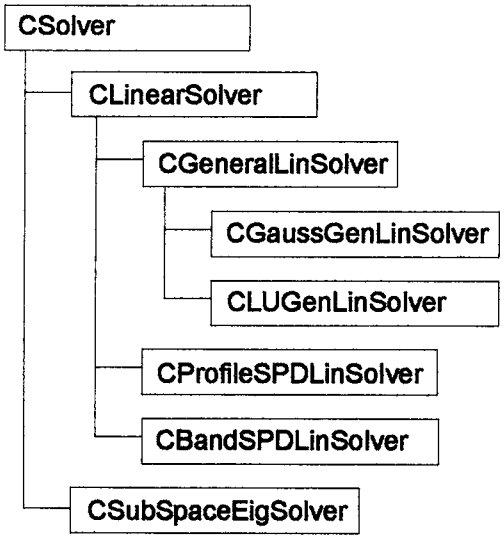


Figure 5.21 Inheritance Tree for Solver Classes

CSolver class is the abstract base class of all solvers. Only member function is the Solve() function, which is defined as a pure virtual function. CLinearSolver is the base class of linear system of equations solvers. It provides functions for setting coefficient matrix and right hand side vectors and getting solution vector. This class is also an abstract base class.

CGeneralLinSolver is the abstract base class for general (unsymmetric or symmetric) system of equations solvers. These solvers stores full coefficient matrix. CGaussGenLinSolver, which is derived from CGeneralLinSolver class, defines a solver that solves a linear system of equations using Gauss elimination method with row pivoting. CLUGenLinSolver class solves a linear system of equations using LU decomposition.

CProfileSPDLinSolver class implements a solver for symmetric positive definite system of linear equations. This class stores only the values inside the profile and above the diagonal of coefficient matrix. Class uses LDL

decomposition to solve the system of equations. This solver can be used for very large system of equations if the equations are numbered such that the profile is minimized.

CBandSPDLinsolver class, which is derived from CLinearSolver, implements a solver for banded symmetric positive definite system of linear equations. Only the upper band of the matrix is stored by the class. Class uses Cholesky factorization to solve the equations. If the bandwidth is small, very large systems can be solved with this solve.

CSubSpaceEigSolver implements an eigen solver. This solver can handle very large generalized eigen problems (for example a system with 10,000 equations) with a diagonal B matrix. Class uses sub-space iteration method to obtain the eigenvalues and eigenvectors.

5.5.8 OpenGL Classes

OpenGL classes are responsible from setting the scene, rendering the model performing object selection operations. There are ten classes in this category. CGL, CScene, CSelectionList and CSelectionManager classes are the most important ones among them. Remaining six classes, which are CLight, CProjection, CSceneState, CSceneVisibility, CSelectedItem and CViewport, are used more like C++ structs. They only have public data members and do not define operations that operate on their data members.

5.5.8.1 CGL Class

This is an abstract base class for the CScene class. Its tasks consist of setting the pixel format, constructing the rendering context and making the

rendering context "current". As it was mentioned in Section 5.2.2, no commands for performing windowing tasks are included in OpenGL. CGL class uses Windows-specific functions to perform its tasks.

5.5.8.2 CScene Class

This class is derived from the CGL class as shown in Figure 5.21. All OpenGL related operations in CALWin are carried out by this class. These includes the initializing the scene, rendering the model, and converting screen coordinates to model coordinates or vice versa.

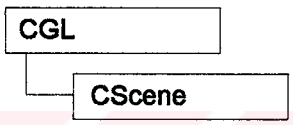


Figure 5.22 Inheritance Diagram for CScene Class

5.5.8.3 CSelectionManager Class

This is a stand-alone class. It implements the object selection mechanism using OpenGL.

5.5.9 Dialog Classes

All dialog boxes in CALWin are managed by corresponding dialog classes. These classes are derived from MFC's CDialog class. There are 43 dialog classes in CALWin. The working principle of these classes and their interfaces are similar. A typical dialog class has a pointer to document object. This pointer is set prior to the displaying of dialog box. Dialog class obtains the data required for its operations from the document object.

5.5.10 Control Classes

Control classes are derived from MFC classes. Each control class defines the behavior of its control. A control is a visual object that can be placed on a dialog or on to application window. There are four control classes in CALWin. These are CTreeBar, COutputBar, CEditStr and CPhantomTree classes.

CTreeBar and CPhantomTree classes define the tree bar that is docked to the left of main application window as shown in Figure 5.4. CPhantomTree is a tree control derived from MFC's CTreeCtrl class. CPhantomTree control fills the client rectangle of CTreeBar which implements the floating bar itself and has nothing to do with the tree control. CTreeBar class is derived from CSizingControlBarG class, which was a third party control class. CEditStr and COutputBar classes define the output bar, which is shown in Figure 5.4. Output bar is used for displaying progress and error messages.

5.5.11 Third Party Controls

There are a number of classes used by CALWin that were not developed in the scope of current study. These classes are not a part of MFC library. For this reason, they are referred as third party controls.

Following third party controls were used in CALWin.

- MFC Grid Control. This control is used as the grid control in matrix editor dialog box. It consists of several classes. This control was written by Maunder, C.

- Sizable Control Bar Control. This control is used for the implementation of tree bar and output bar. This control consists of several classes. It was written by Posea, C.
- XGraph Chart Control. It is used for time-history plots. It consists of several classes. Unlike other controls this control is dynamically linked to the CALWin.
- Flat button control. This control is used in several dialog boxes. It was written by Calabro, D.

CHAPTER 6

NUMERICAL EXAMPLES AND COMMENTS

In this chapter five different problems will be solved using CALWin. The solution is presented in a step-by-step manner with screenshots so that the reader can get an idea and feeling about the running of CALWin.

6.1 Frequencies and Mode Shapes of Beam on Elastic Foundation

In this problem, frequencies and mode shapes of a beam resting on a two-parameter elastic foundation with both ends fixed are evaluated. This problem is taken from the studies of Alemdar (1995).

6.1.1 Problem Data

The configuration of the beam is shown in Figure 6.1.

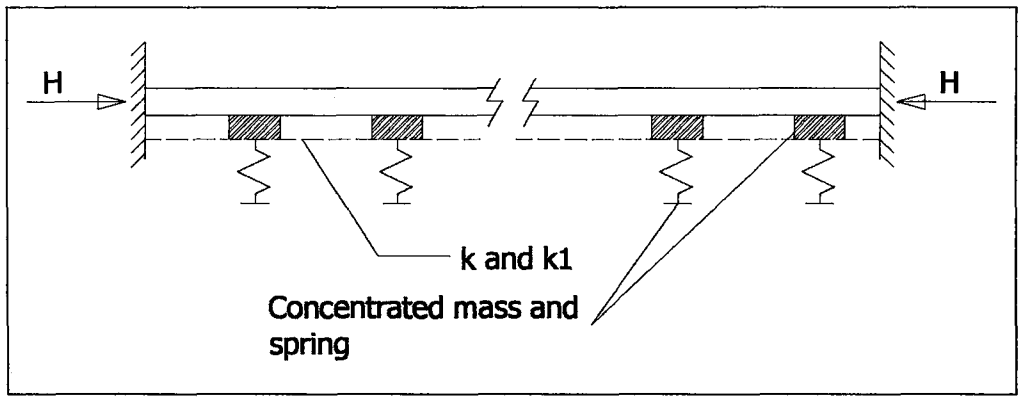


Figure 6.1 Beam on Elastic Foundation

The input data for the problem is as follows. Foundation parameters k and k_1 are 2500 kN/m^2 and 2000 kN, respectively. Concentrated mass of 125 kg and a translational spring with a stiffness of 5000 kN/m are attached per unit length. Remaining parameters of the beam are:

$EI = 5000$ kNm^2

$m = 60$ kg/m

$H = 500$ kN

$L = 35$ m.

Beam is divided into 35 segments. Vibration frequencies of the beam are obtained from the following relation:

$$\left(\mathbf{K}_{sys} - \mathbf{G}_{sys}\right) - \omega^2 \mathbf{M}_{sys} = 0 \qquad (6.1)$$

where $\mathbf{K}_{sys}$, $\mathbf{G}_{sys}$, and $\mathbf{M}_{sys}$ are assembled system stiffness, geometric stiffness and mass matrices. Values of ω^2 are the eigenvalues of the above equation. Problem is solved for both Winkler type foundation and two-parameter foundation.

6.1.2 Analysis in CALWin

The first step in the analysis of a structure in CALWin is the setting of the model geometry. However, it is also possible to analyze a given structure without defining it in the graphic editor.

The grid system must be defined in proper coordinates before starting to enter the structure in graphic editor. Grid system is set up through a dialog box. This dialog box is displayed selecting *Grid settings* from the *File* menu or by

double-clicking a grid line in the graphic editor. In both cases, the dialog box shown in Figure 6.2 is displayed.

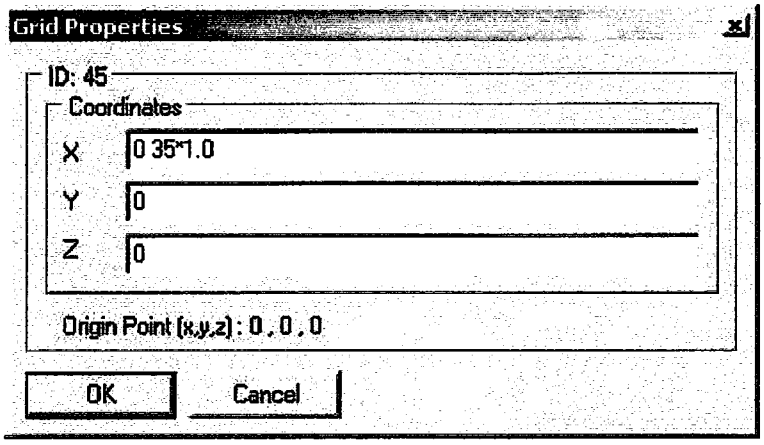


Figure 6.2 Grid Settings Dialog

The first number in any one of three edit boxes is the start coordinate of the first grid line on the specified axis. Then, the number of grid lines multiplied by the grid spacing is specified. Since the beam we are analyzing is 35 m long, we divide in 35 pieced each with 1 meter long. The graphic editor looks like as shown in Figure 6.3 after dismissing the dialog with OK button.

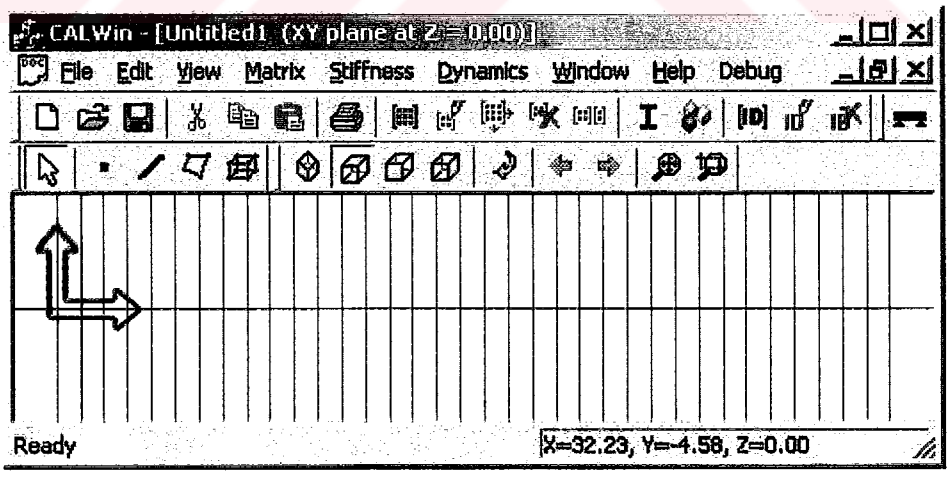


Figure 6.3 Grid System for Beam on Elastic Foundation

Next the beam is defined with line elements. Line element represents a one-dimensional finite element. In general, truss, frame, beam on elastic foundation or pile elements are all defined by line element command. This

command is invoked by a toolbar button (see Figure 6.4). Beam is defined as shown in Figure 6.4 using the mouse and grid snap points.

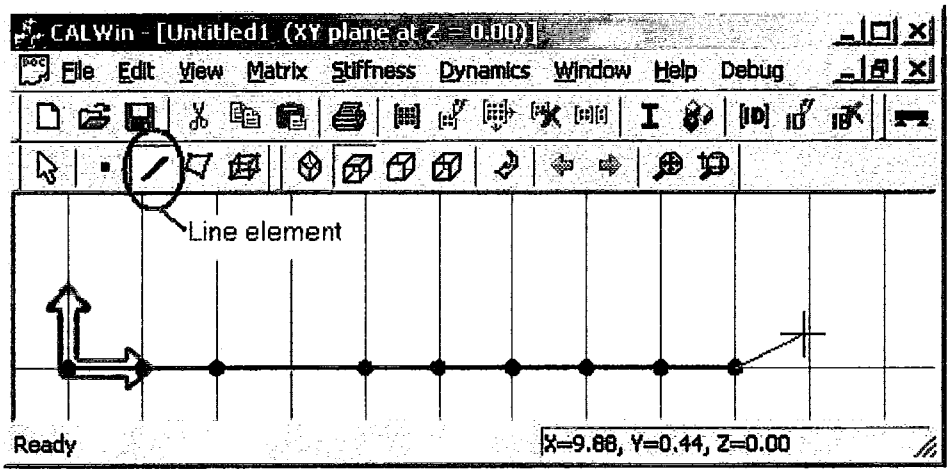


Figure 6.4 Defining the Line Elements

Next step is the definition of the material and cross-section of the structural element, a beam resting on the foundation in this case. Materials and sections are defined using the material editor and section editor, respectively. There are various types of elements CALWin and each requires different number of cross-sectional and material properties. For example, while truss element needs only the area of the section, three-dimensional frame element needs moment of inertias about two axes, torsional constant and area of the section. It is sufficient to define the only necessary parameters in material and section editors. Material and section editors are invoked from the toolbar or by double clicking on the icons of already defined materials and sections. Material property for the current problem is the modulus of elasticity, E=5000 kN/m^2 (assuming that I = 1 m^4), as shown in Figure 6.5.

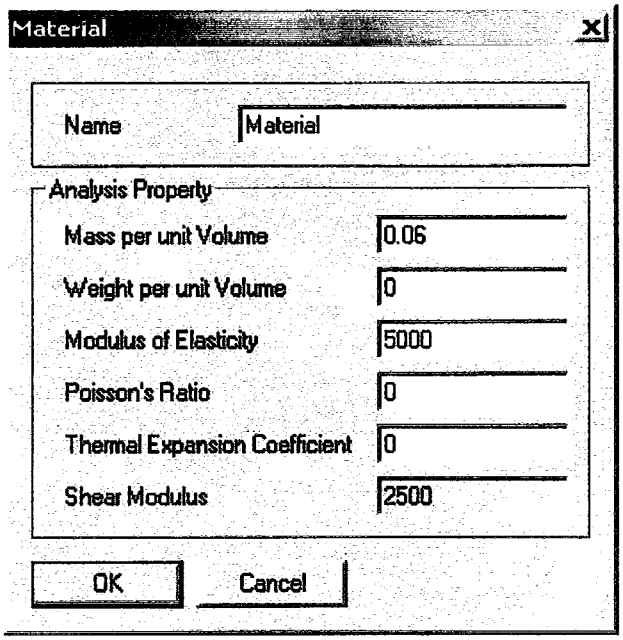


Figure 6.5 Material Settings of the Beam

Once the material and sections are defined, they must be assigned to the line elements that have been defined. This is done by line element property editor. This, property editor is a dialog box in which the attributes and properties of the element are shown. Property editor is shown in Figure 6.6. This dialog box is accessed by double clicking the desired element. After assigning a section and material two all elements, we are ready to calculate the element matrices of beam segments. Since all segments are identical, we only need to calculate stiffness, geometric stiffness and mass matrices for one segment. This operation is also carried out through the line element property editor. Once again, we open the property editor by double clicking any element. Then we select the element type as BEF (Beam on Elastic Foundation, which is a Winkler type foundation) from the element type combo box (see Figure 6.6) and click to create element matrices. Another dialog box is displayed as shown in Figure 6.7. This dialog box is also accessible from the toolbar and from the "Stiffness -> Elements" menu item.

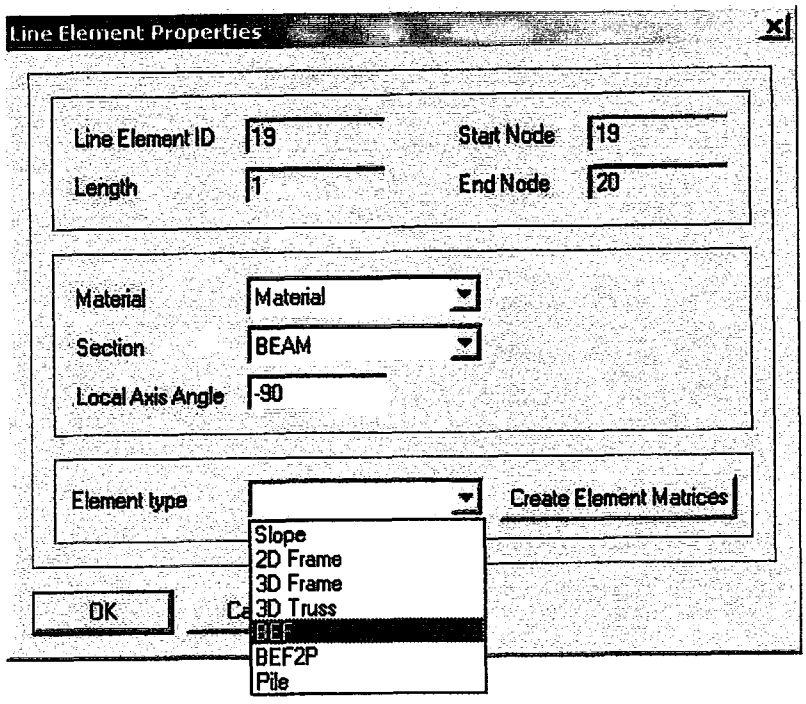


Figure 6.6 Line Element Property Editor

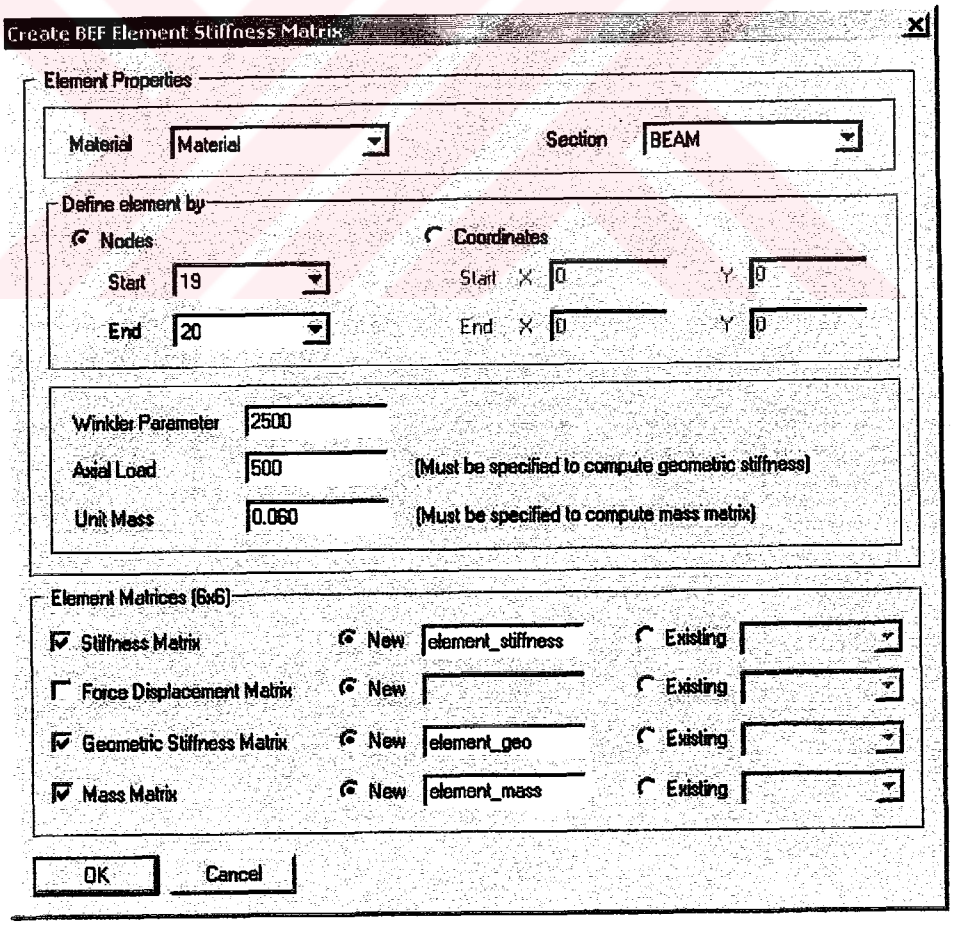


Figure 6.7 Creation of Element Matrices

Material and section properties and node numbers are selected automatically, if the dialog box is displayed from the element property editor. Otherwise, these parameters should also be selected by the user. The remaining parameters are the Winkler parameter, compressive axial load and mass per unit length of the element. Axial load and unit mass are optional parameters and they have to be specified only if the geometric stiffness matrix and mass matrix is to be created. In the bottom part of the dialog box, the output options are specified. The matrices that are to be created are selected by checking the appropriate check boxes. In Figure 6.7, only the force-displacement matrix is not created. It is possible to store the computed element matrices in already defined (existing) matrices or in new matrices (which need to be created automatically). In this case, we want to create new matrices, and specify their names as shown in Figure 6.7. As we click OK and matrices are generated and displayed on the tree view (see Figure 6.8).

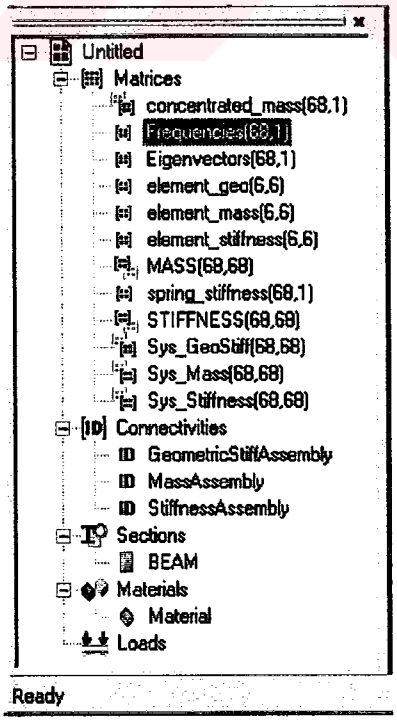
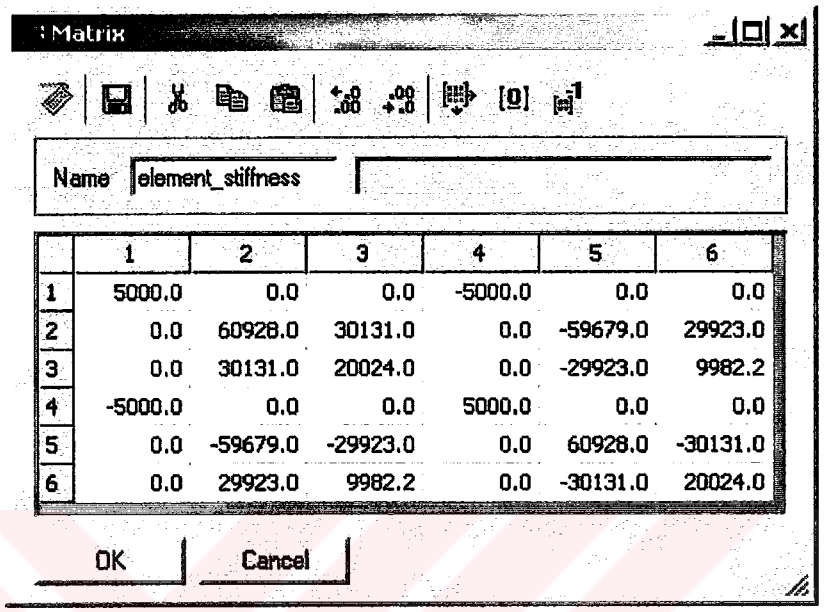


Figure 6.8 All Matrices and Attributes Defined in the Model

Double clicking on to the icon of a matrix, opens the matrix editor and displays the matrix in a grid. The stiffness matrix of the beam on elastic foundation is shown in Figure 6.9.



	1	2	3	4	5	6
1	5000.0	0.0	0.0	-5000.0	0.0	0.0
2	0.0	60928.0	30131.0	0.0	-59679.0	29923.0
3	0.0	30131.0	20024.0	0.0	-29923.0	9982.2
4	-5000.0	0.0	0.0	5000.0	0.0	0.0
5	0.0	-59679.0	-29923.0	0.0	60928.0	-30131.0
6	0.0	29923.0	9982.2	0.0	-30131.0	20024.0

Figure 6.9 Stiffness Matrix of a Typical Segment of the Beam

Matrix editor can be used to view automatically generated matrices, to modify them, or to edit an empty matrix. Using the matrix editor, it is possible to invert the matrix (if the inverse exists), resize it, write it to a file or copy to clipboard and from there to other applications, like in Excel.

Now it is time to number the degrees of freedom. There are 34 translational and 34 rotational degrees of freedom. Axial degrees of freedom are not considered in the analysis. Degrees of freedom in CALWin are specified in a special matrix, which is called ID matrix. The rows of ID matrices designate the elements and columns are used for degrees of freedom. Last column of an ID matrix holds the names of the matrices to be assembled for each line. ID matrices are edited using an editor similar to matrix editor (see Figure 6.10).

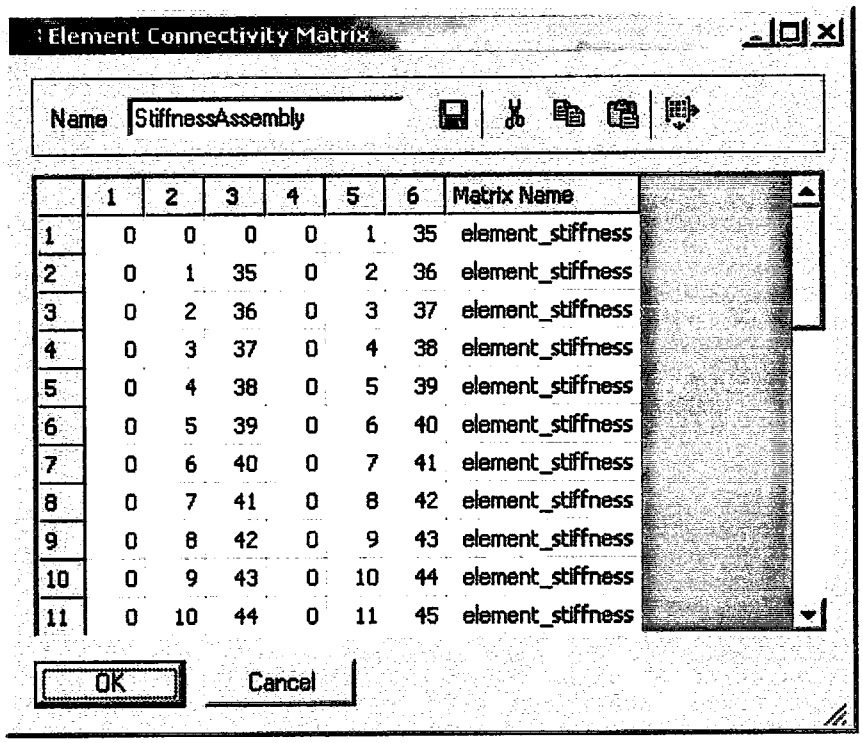


Figure 6.10 Connectivity Array for the Element Stiffness Matrices

Degrees of freedom are numbered such that first 34 degrees of freedom are translational which simplifies the drawing of mode shapes later.

Element matrices are assembled selecting the *Assemble Element Matrices* command from the *Stiffness* menu. This command displays a dialog as shown in Figure 6.11.

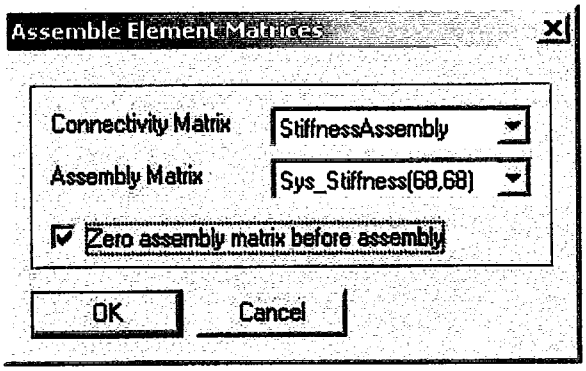


Figure 6.11 Assembly of Element Stiffness Matrices to System Stiffness Matrix

Using this command we assemble the element stiffness matrices, element geometric stiffness matrices and mass matrices system matrices, which should be defined before this step using the *Create New Matrix* command from the *Matrix* menu (see Figure 6.8).

From equations 6.1, we know that, in order to compute total stiffness matrix, the geometric stiffness matrix is subtracted from the stiffness matrix. We perform this operation using the Matrix Operation dialog, which can be open by *Define Matrix Operation* item from the *Matrix* menu. Matrix operation dialog is shown in Figure 6.12. With the help of this dialog all matrix operations summarized in Table 5.1 can be performed. We select the subtraction operation, and *Sys_Stiffness* and *Sys_GeoStiffness* matrices as the argument matrices. The result is stored in a new matrix named *STIFNESS*.

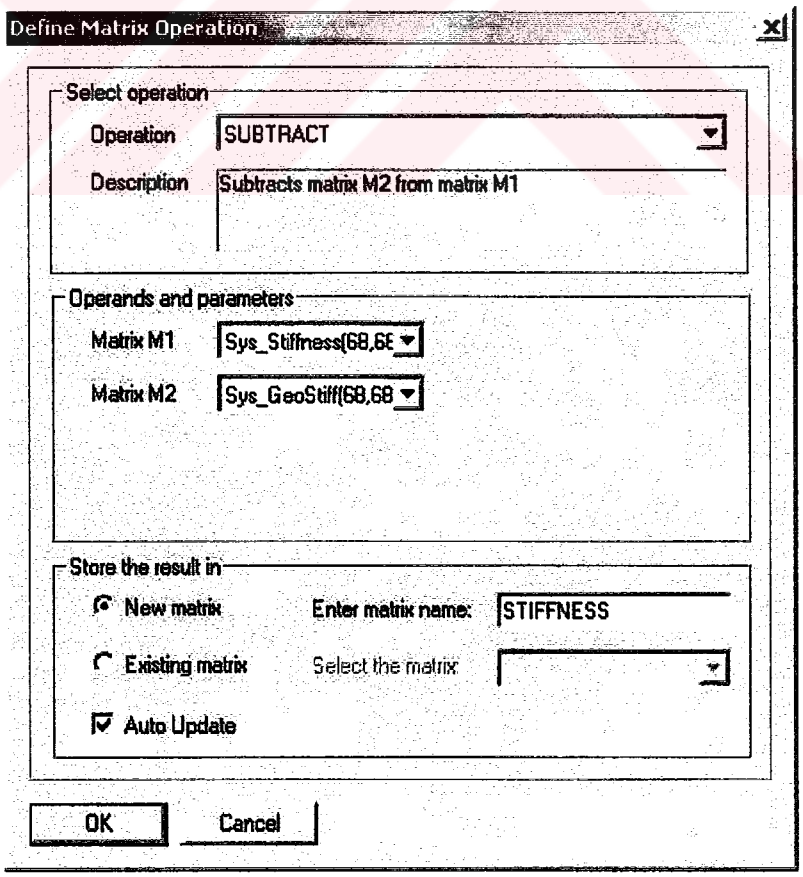


Figure 6.12 Computation of Total System Stiffness Matrix

We should add the contributions of concentrated masses and springs to system mass and stiffness matrices, respectively. We create two column vectors named *concentrated_mass* and *spring_stiffness* (see Figure 6.8). Springs and masses correspond the translational degrees of freedom. Hence, only first 34 for terms (corresponding to translational degrees of freedom) of these column vectors are non-zero and remaining terms are zero. These vectors should be added to the diagonals of the total system stiffness matrix and mass matrix (see Figure 6.13).

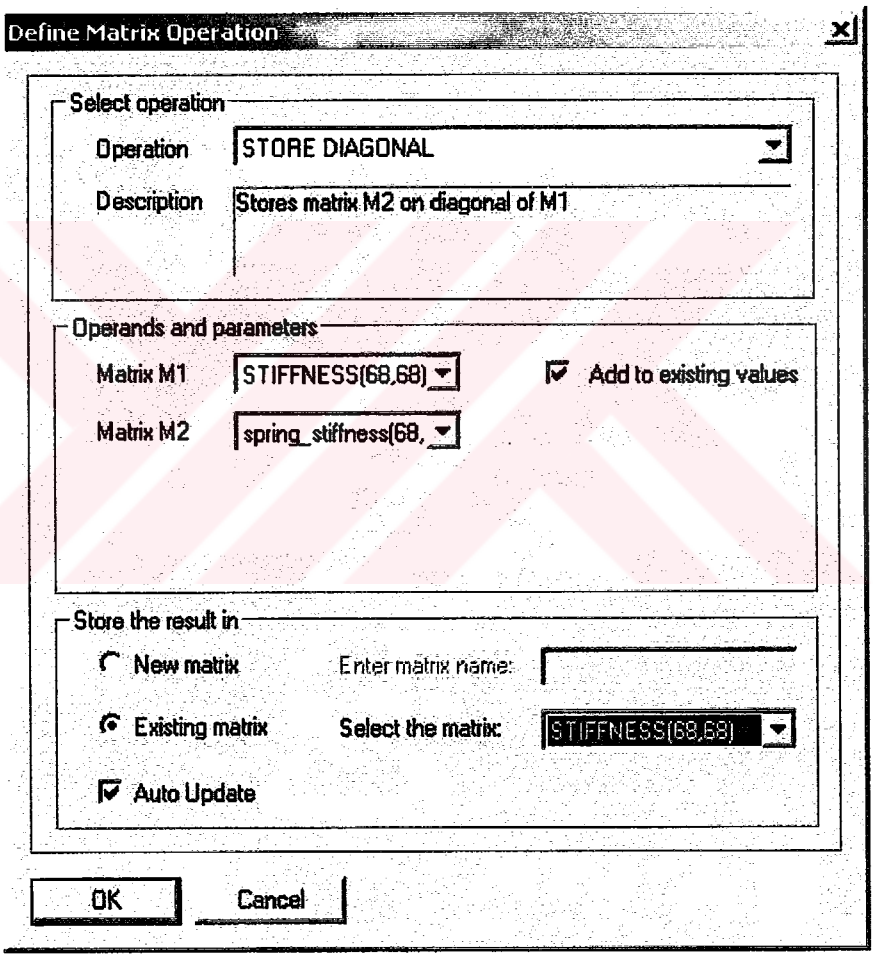


Figure 6.13 Addition of Spring Stiffness to the Total Stiffness

The last step is the calculation of the eigenvalues of the Equation 6.1. Eigenvalues are computed using the Jacobi method. We choose the *Jacobi* command from the *Dynamics* menu. The input and output parameter for this command are specified through a dialog box which is shown in Figure 6.14

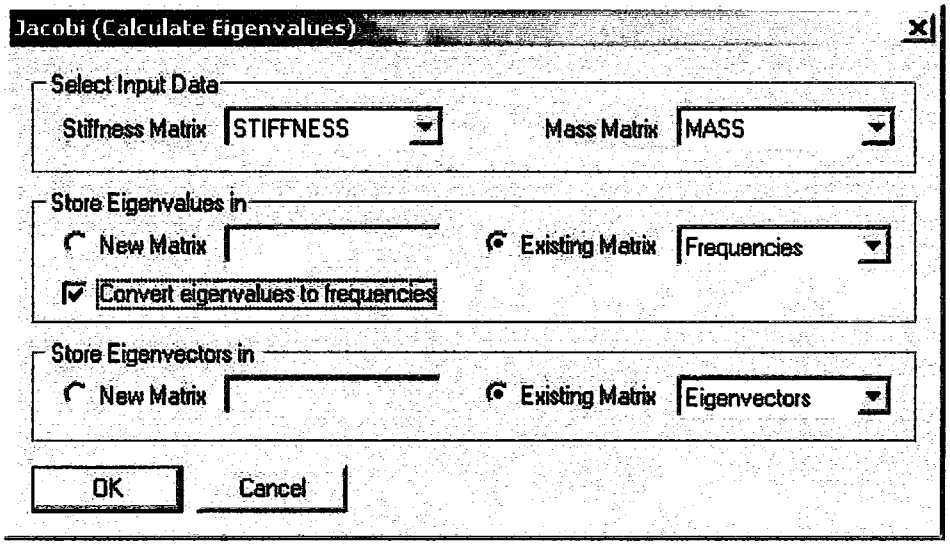


Figure 6.14 Computation of Eigenvalues

The Jacobi command computes the eigenvalues of the generalized eigenvalue problem, which is given by the equation:

$$\mathbf{KV} = \omega^2 \mathbf{MV} \tag{6.2}$$

The input parameters are the stiffness matrix and the mass matrix. Therefore, the total stiffness matrix and mass matrix are selected using the combo boxes. For output, formerly created column vector *Frequencies* and matrix *Eigenvectors* are selected. However, it is also possible to store the outputs in new matrices. There is one more option in this dialog box: conversion of eigenvalues to frequencies by taking square root of each item, if desired. This option is selected, because we want the frequencies, not the eigenvalues.

Same operations are repeated for the two-parameter foundation case. All steps are identical except that, the element type of all elements is BEF2P instead of BEF.

6.1.3 Results

The lowest four vibration frequencies are shown in Table 6.1. As seen from the results, vibration frequencies of the two-parameter case are slightly higher. Comparison of these results with the studies of Alemdar (1995) shows that they are identical.

Table 6.1 Vibration Frequencies (rad/s)

	ω_1	ω_2	ω_3	ω_4
Winkler foundation	201.28	201.31	201.49	202.22
Two-parameter foundation	201.58	202.26	203.61	209.52

The plots of mode shapes corresponding to lowest four frequencies for both type of foundation are shown in Figures 6.15 to 6.28.

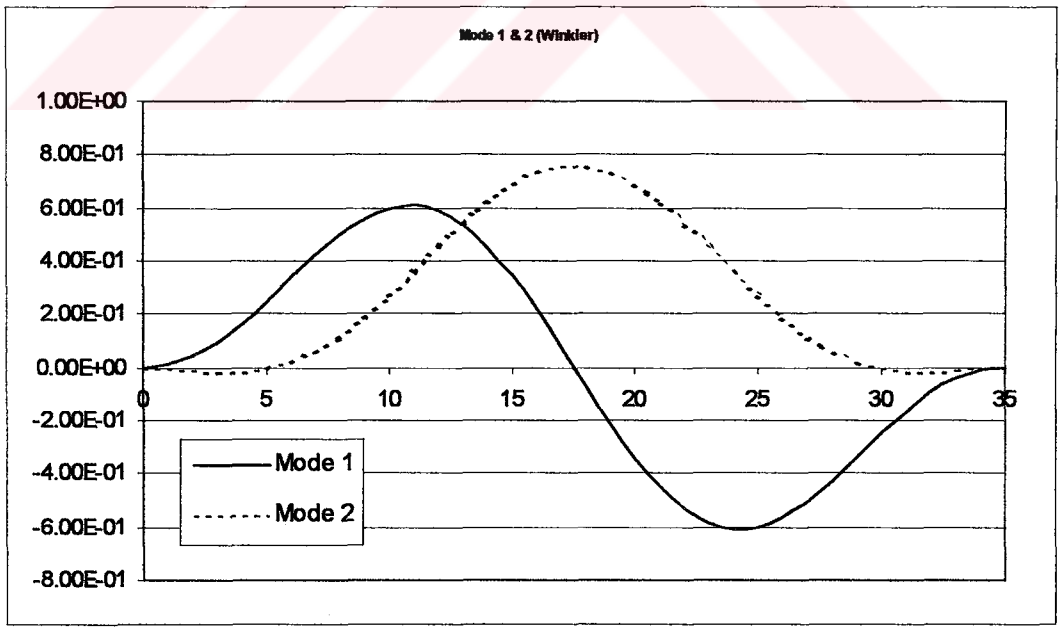


Figure 6.15 Mode 1 and 2 for Beam on Winkler Foundation

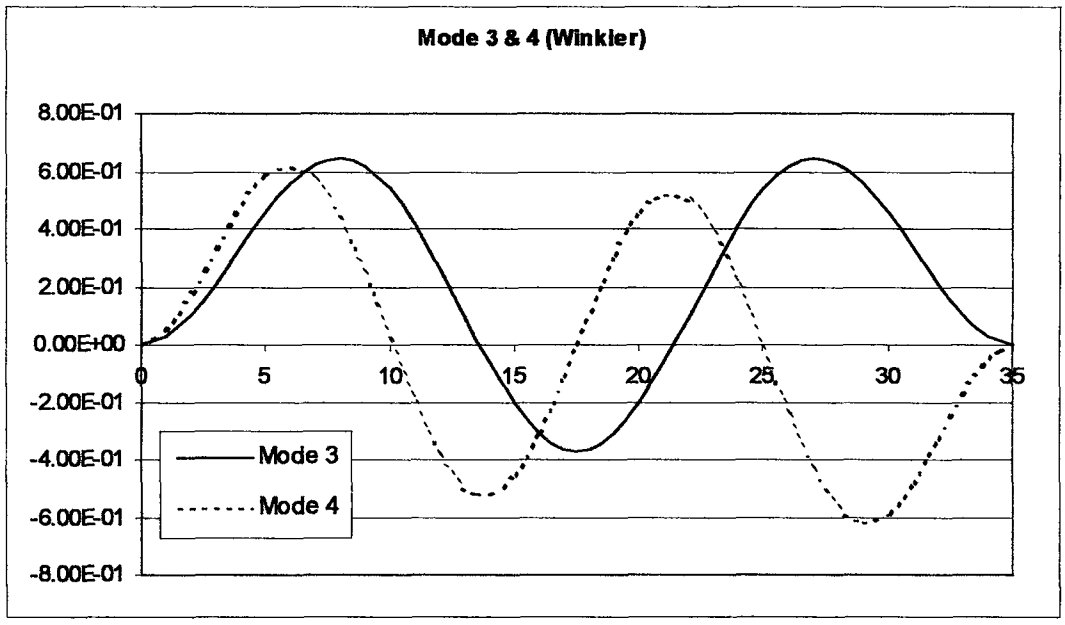


Figure 6.16 Mode 3 and 4 for Beam on Winkler Foundation

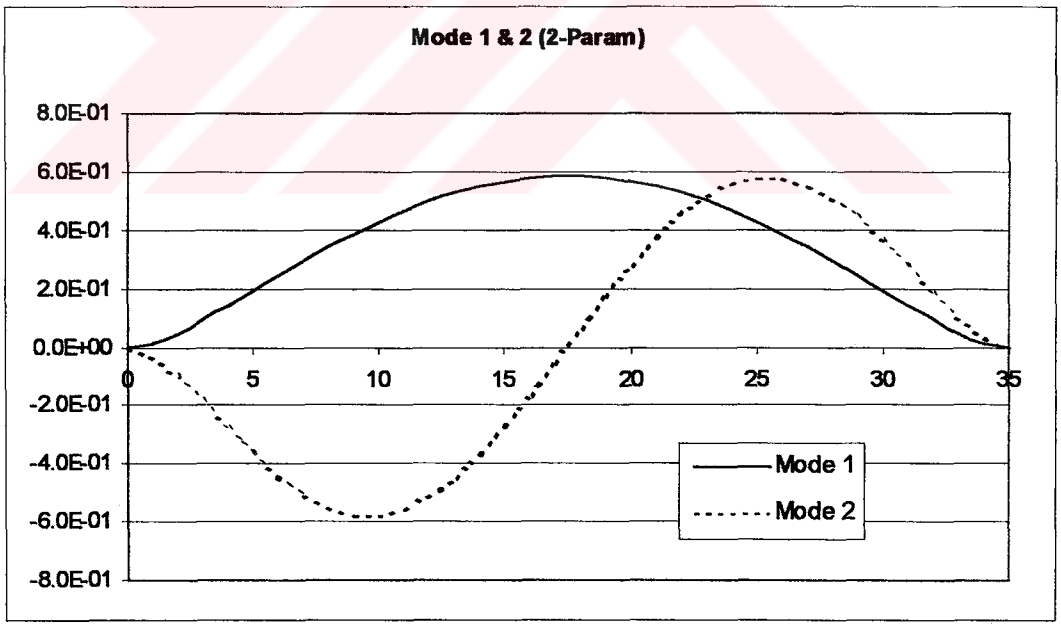


Figure 6.17 Mode 1 & 2 for Beam on 2-Parameter Foundation

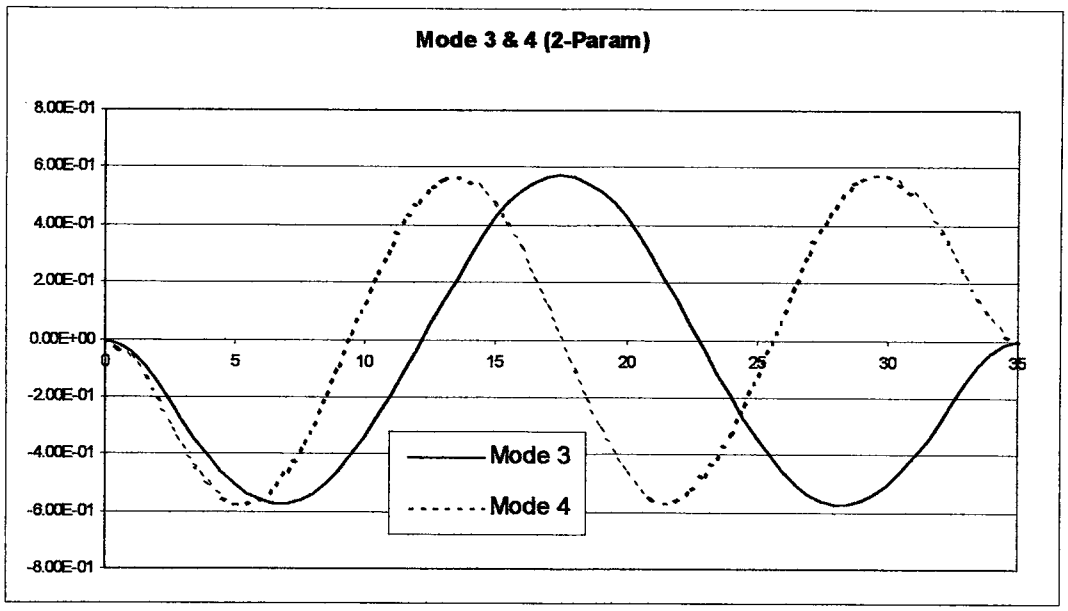


Figure 6.18 Mode 3 & 4 for Beam on 2-Parameter Foundation

6.2 Dynamic Analysis of a Frame Structure

In this section, dynamic analysis of a frame structure using CALWin is described. The steps are identical with the steps of previous problem considering the evaluation of frequencies and mode shapes.

6.2.1 Problem Data

The structure to be analyzed is shown in Figure 6.19. Structure is a 10-story, 2-bay plane frame. Story height is 3.6 meters and is same for all stories. A lateral load with a time variation is applied to the structure. Additional masses of 10.5 t exist at every floor level. Damping ratio is 5 percent for all modes. All girders have the same cross-section. There exist 4 types of columns, which are indicated in Figure 6.19 by C1, C2, C3, and C4. Girders are indicated by the symbol G. The modulus of elasticity is 2.0E8 kN/m^2. Section properties of the columns and girders are listed in Table 6.2. The time variation of the applied load is shown in Figure 6.20.

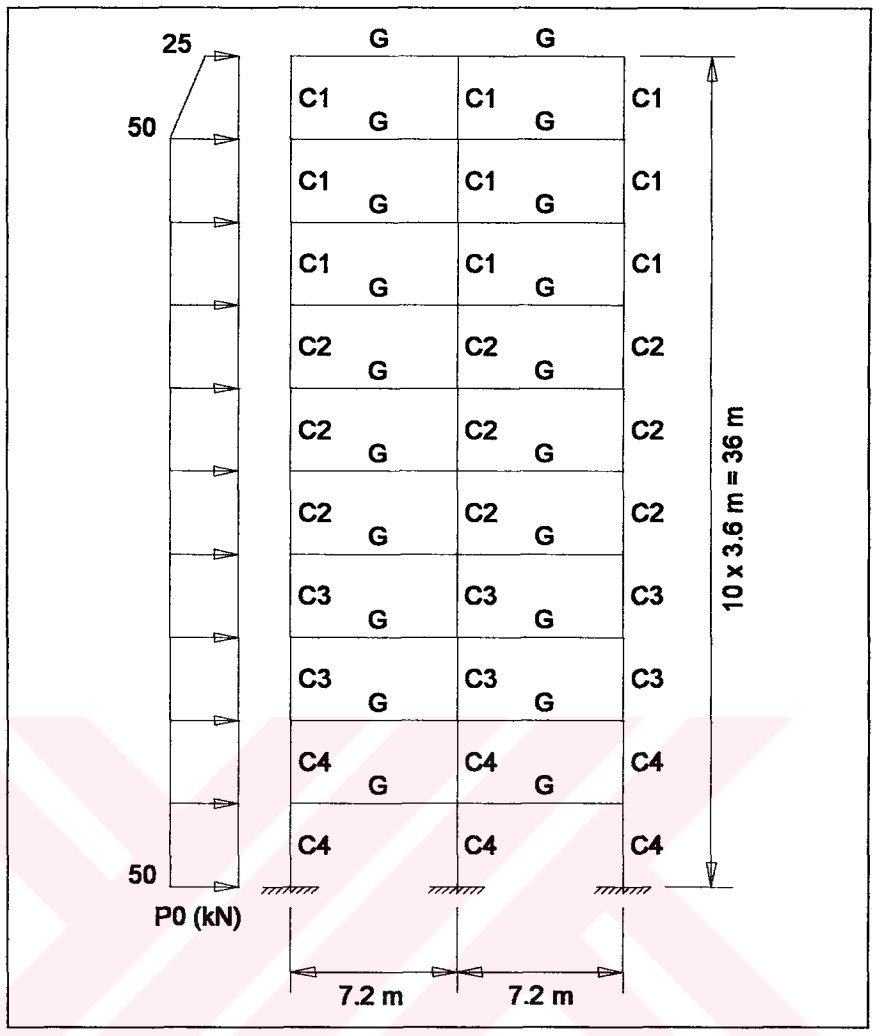


Figure 6.19 10-Story Frame Structure

Table 6.2 Section Properties of the Members

	I (m^4)	m (kg/m)
G	4.75E-4	840
C1	4.57E-5	460
C2	1.43E-4	900
C3	3.01E-4	1280
C4	3.88E-4	1600

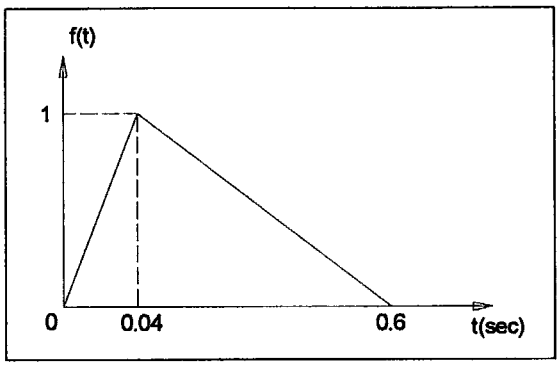


Figure 6.20 Time Variation of Applied Load

6.2.2 Analysis in CALWin

The steps for defining the geometry of the structure are identical to those described in previous example. Firstly, grid system is set up. Then the structure is drawn in graphic editor. Material and frame sections are defined using material editor and section editor. Since these steps are explained in the previous example in detail, they are not repeated here. After performing these tasks, application window looks like as shown in Figure 6.21.

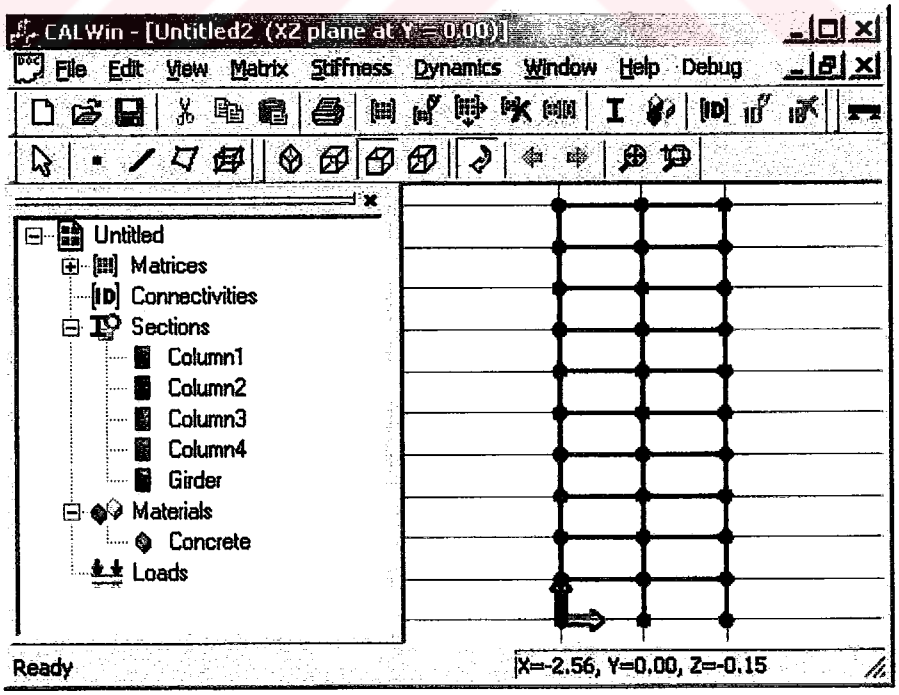


Figure 6.21 Frame Structure Defined in Graphic Editor

Element stiffness matrices are created as shown in Figure 6.6. However, this time, element type is *Slope* since the axial deformations in beams and columns are neglected. For each element type, that is for columns C1, C2, C3 and C4 and for girder a 4x4 element stiffness matrix is created. Consistent mass matrices are created for all elements. This is done manually, in other words, using the matrix editor. The consistent mass matrix of the girders is shown in Figure 6.22.

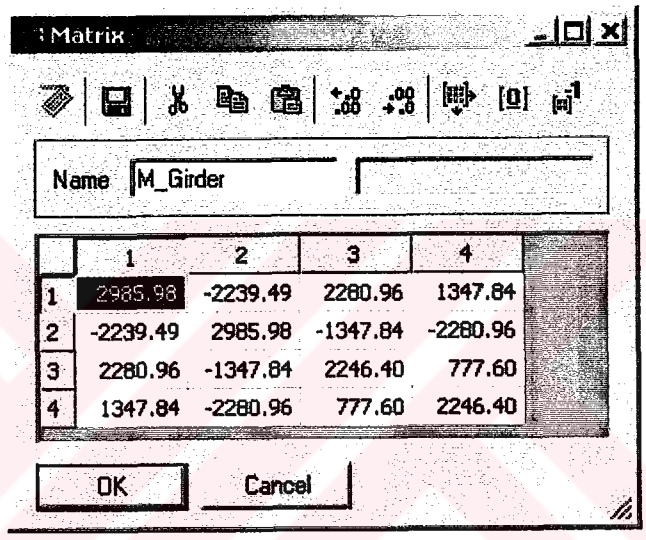


Figure 6.22 Consistent Mass Matrix for the Girders

The numbering of degrees of freedom is such that, translational degrees of freedom of floors are given first ten numbers. Remaining degrees of freedom are all rotations.

System stiffness and mass matrices are created and assembled as illustrated in previous example using the connectivity matrix editor. Next, rotational degrees of freedom are statically condensed out from both system stiffness and mass matrices. Additional story masses are added to condensed mass matrix. Evaluation of eigenvalues and eigenvectors is identical to the steps described in previous chapter. Thus, it is not repeated here.

Having determined the frequencies and mode shapes, uncoupled dynamic response of the structure can now be determined. Firstly, the damping matrix as column vector and the load vector are defined using the matrix editor. Then, the time function is defined using a special matrix. This matrix should have only two columns. Number of rows depends on the number of entries on the time axis. This matrix is shown in figure 6.23.

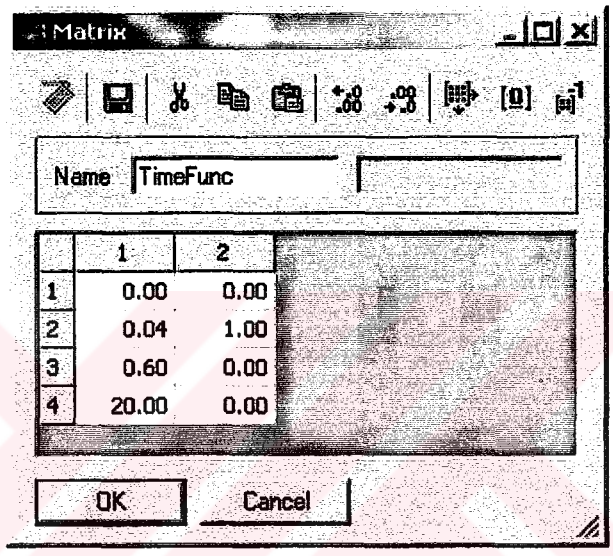


Figure 6.23 Time Variation of the Applied Load

Dynamic analysis is performed using the *Uncoupled Dynamic Response* command, which is available in *Dynamics* menu. This command needs six input parameters. These are the angular frequencies as a column vector, diagonal damping matrix as a column vector, forcing function as a column vector, time function as described above, number of time points and time increment. Number of time points and time increment determines the time scale at which the principle coordinates are generated. Output of the function is the principle coordinates of the mode shapes. These variables and parameters are defined in the Uncoupled Dynamic Response dialog, which is shown in Figure 6.24.

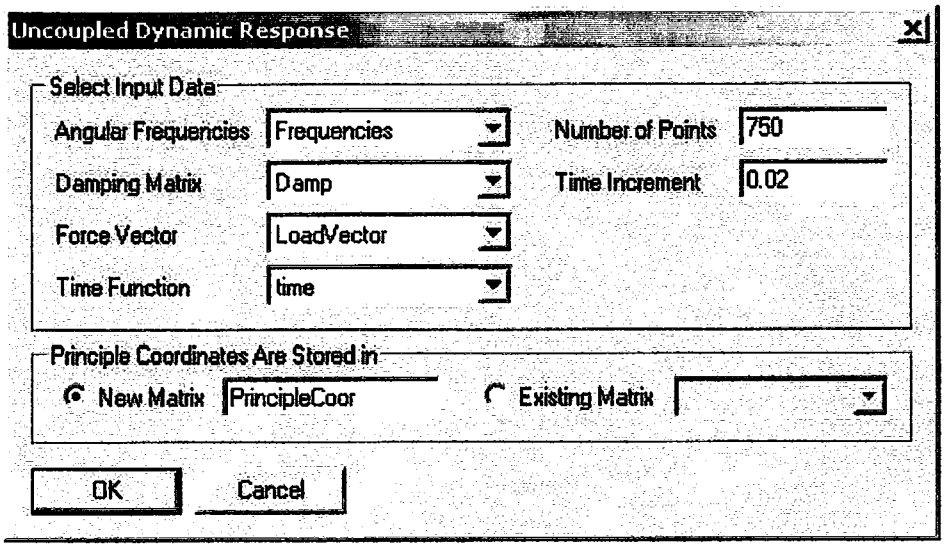


Figure 6.24 Uncoupled Dynamic Response

The physical displacements are calculated by premultiplying principle coordinates with the eigenvectors. This is accomplished using the matrix operation dialog.

6.2.3 Results

The frequencies and periods of the structure shown in Figure 6.19 are presented in the Table 6.3. In Figure 6.25, time-history plot of the top story displacement is shown.

Table 6.3 Frequencies and Period

Mode	ω (rad/s)	T (sec)
1	4.375	1.4367
2	10.485	0.59925
3	18.031	0.34846
4	26.018	0.24149
5	33.869	0.18551

Table 6.3 (continued) Frequencies and Periods

Mode	ω (rad/s)	T (sec)
6	40.791	0.15403
7	47.792	0.13147
8	62.773	0.10009
9	74.726	0.84084E-1
10	96.734	0.64953E-1

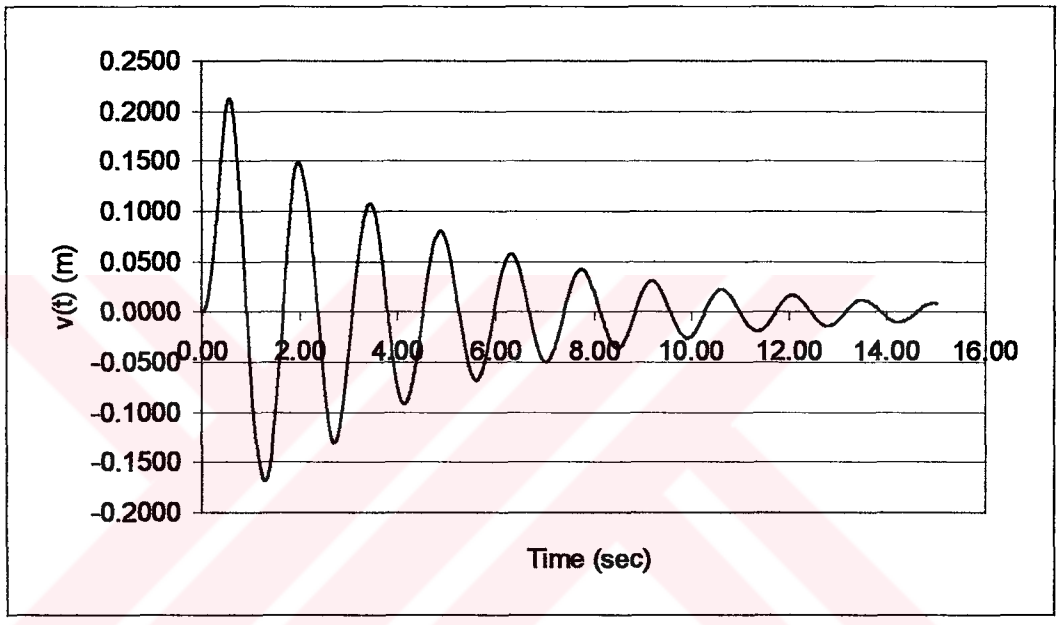


Figure 6.25 Top Story Displacement History

6.3 Buckling Load Analysis of a Column

Critical buckling load of a column can be evaluated by solving the following set of equations:

$$\mathbf{ku} - \lambda_G \mathbf{k}_G \mathbf{u} = \mathbf{0} \tag{6.3}$$

Here, $\mathbf{k}$ is the stiffness matrix of the column and $\mathbf{k}_G$ is the geometric stiffness matrix when a unit axial load is applied. $\mathbf{u}$ is the displacement vector and λ_G is the axial load factor.

Solution to this equation is obtained by solving the eigenvalue problem in Equation 6.4.

$$\|\mathbf{k} - \lambda_G \mathbf{k}_G\| = 0 \tag{6.4}$$

Eigenvalues represent the values of the axial-load factor λ_G at which the buckling occur and eigenvectors represent the buckling mode shapes. In practice only the first buckling load factor and mode shape have a significance since the column will fails when the axial load exceeds the first critical load.

6.3.1 Problem Data

Critical buckling load of the column shown in Figure 6.26 is inspected for two different support conditions. First one is hinged at bottom and hinged at top with vertical deflection released. Second condition is fixed at the bottom and top with vertical deflection released.

Column is a 50 cm by 50 cm concrete column with E=2.5 x 10^7 kN/m^2 and I = 5.208 x 10^{-3} m^4. Height of the column, denoted by L, is 4 m. Column is divided into 25 equal pieces.

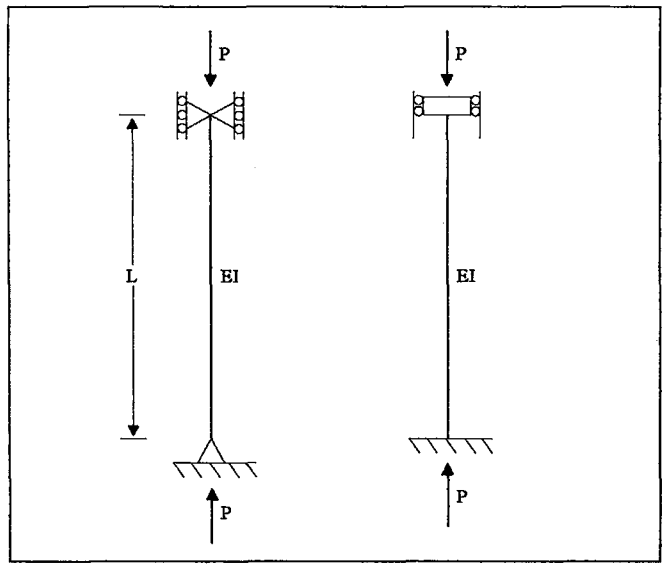


Figure 6.26 Column With Hinged and Fixed Support Conditions

6.3.2 Analysis in CALWin

Modeling starts with setting the grid system. Then the column is drawn in the graphic editor. After that, material and frame sections are defined as in the first example. At this step, application window looks like as shown in Figure 6.27.

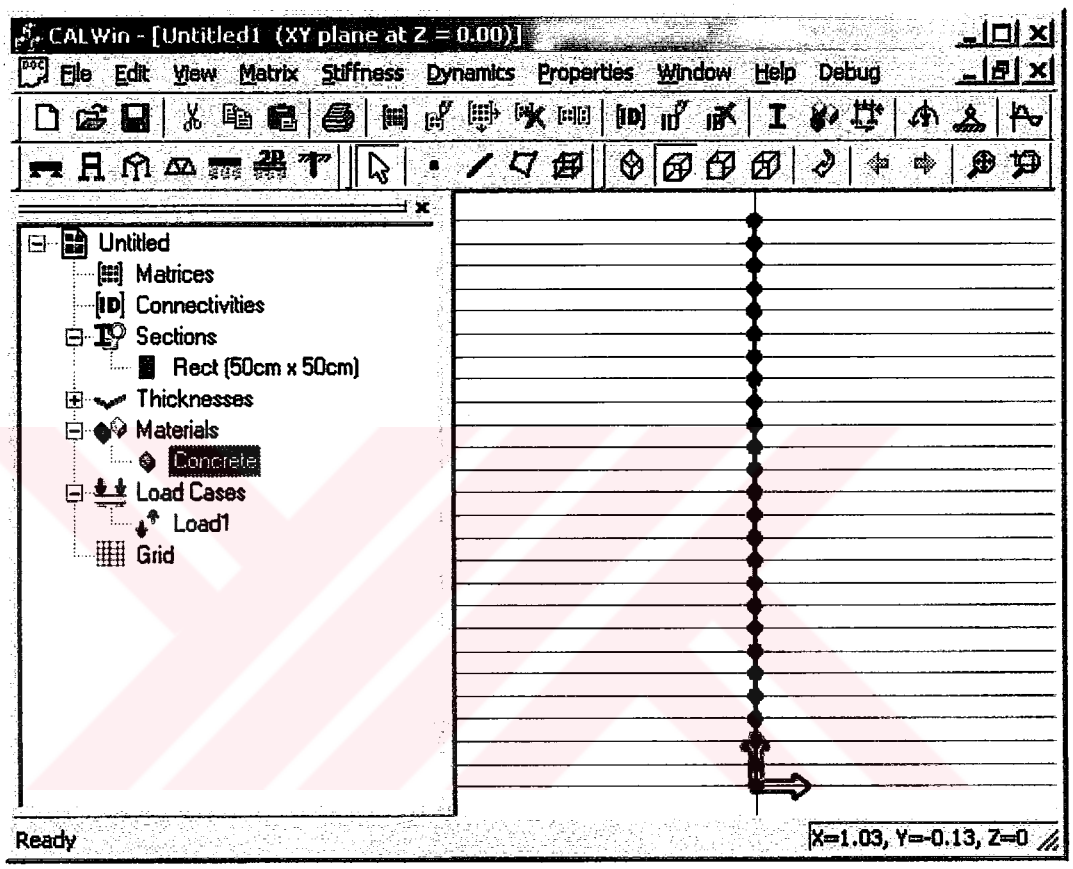


Figure 6.27 Column Defined in Graphic Editor

Next step is the creation of element stiffness matrix. This is done by double clicking on an element in the graphic editor and pushing *Create Element Matrices* button in the line element property editor, which is shown in Figure 6.6. Element type is selected as slope. In the element creation dialog box, a new matrix is created with a name *ElementStiffness*. This is shown in Figure 6.28. Geometric stiffness matrix of the slope element created manually using the matrix definition given in Equation 6.5.

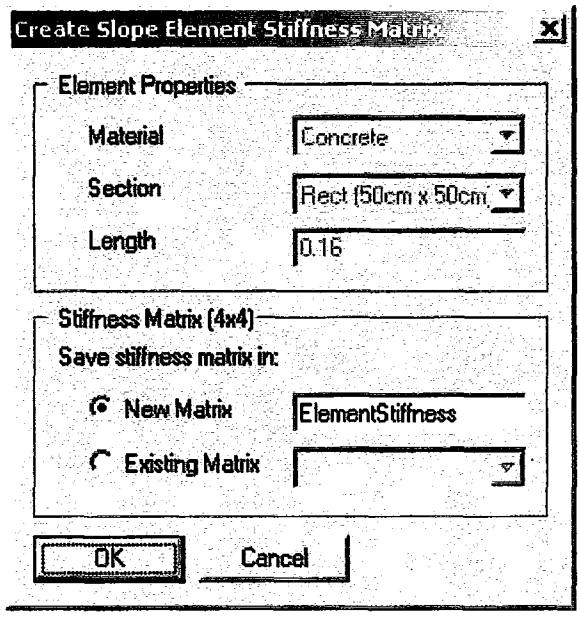
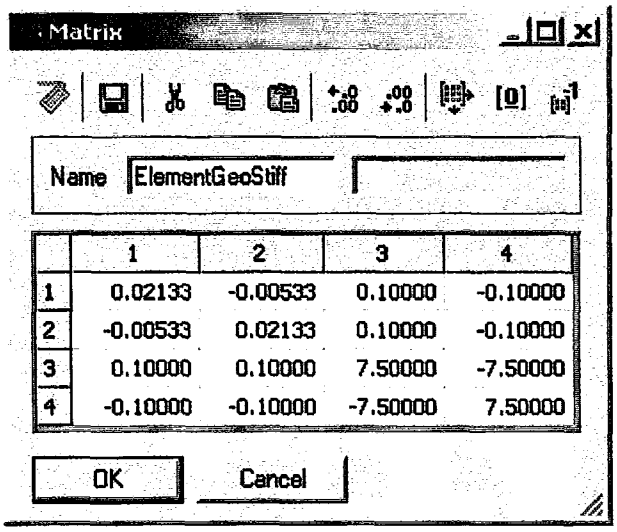


Figure 6.28 Creation of Element Stiffness Matrix

$$\frac{1}{30L}\begin{bmatrix} 4L^2 & -L^2 & 3L & -3L \\ -L^2 & 4L^2 & 3L & -3L \\ 3L & 3L & 36 & -36 \\ -3L & -3L & -36 & 36 \end{bmatrix} \tag{6.5}$$

Elements of this geometric stiffness matrix are ordered according to the ordering of degrees of freedom of Slope element. First two degrees of freedom correspond to rotations at ends and the last two correspond to lateral deflections at ends. Calculated geometric stiffness is shown in Figure 6.29.



	1	2	3	4
1	0.02133	-0.00533	0.10000	-0.10000
2	-0.00533	0.02133	0.10000	-0.10000
3	0.10000	0.10000	7.50000	-7.50000
4	-0.10000	-0.10000	-7.50000	7.50000

Figure 6.29 Element Geometric Stiffness Matrix

Degrees of freedom are numbered such that, translational degrees of freedom are given first 24 numbers. Remaining degrees of freedom are all rotations.

Since the numbering is manual, we create element connectivity matrices. Four element connectivity matrices are created. Two of them are for the hinged system (one for assembly of structural stiffness matrix and one for assemble of structural geometric stiffness matrix) and the other two are for the fixed system.

Eigenvalues and eigenvectors are evaluated as explained in the first example. Thus, the process is not repeated here. Eigenvalues and eigenvectors for both systems are stored in new matrices. These matrices are *BucklingLoads_Fix, BucklingLoads_Hinge, Shapes_Fix* and *Shape_Hinge* as shown in Figure 6.30.

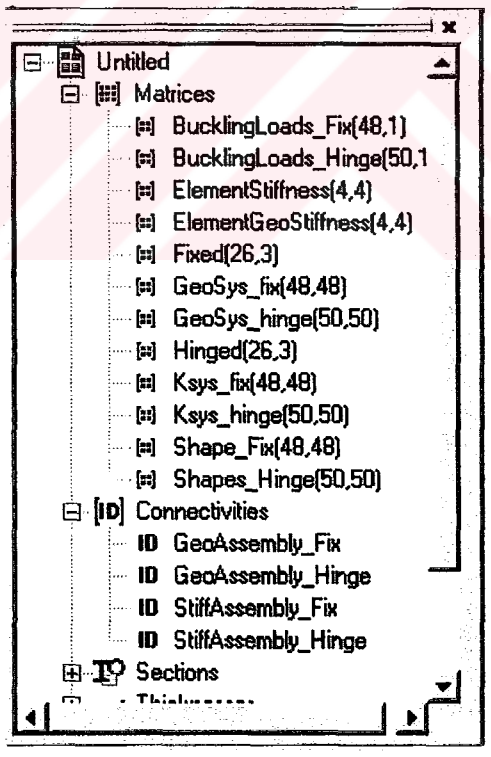


Figure 6.30 Matrices Created for the Buckling Analysis

In order to view buckling mode shapes in the program, first 24 rows of the first three mode shapes are copied to new matrices named *Fixed* and *Hinged* for

the fixed and hinged systems, respectively. Rotational degrees of freedom are not used to display the buckling mode shapes. Then these matrices are displayed in graph view. To achieve this, first the Time History command is invoked from the toolbar. This displays the following dialog box.

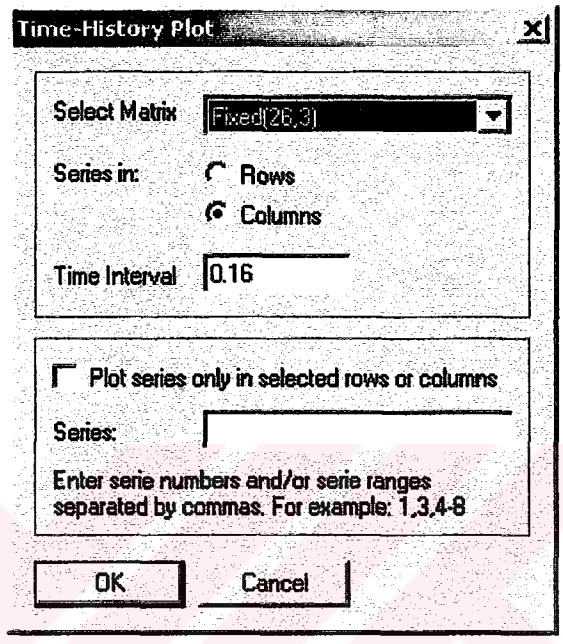


Figure 6.31 Time-History Plot Dialog

The dialog box shown in Figure 6.31 is actually used for displaying time-history of response quantities; however, in this case it can be utilized for viewing the mode shapes. The matrix that is to be plotted is selected in the combo box, which is the matrix containing first three mode shapes for the hinged column. Then, the "Series in Columns" option is selected because the shapes are stored in columns for the current problem. The length of each element, which is 0.16 m, is entered to the time interval edit box. This does not change anything other than the number displayed in time-axis actually. Closing the dialog box with OK button displays a graph view containing the mode shapes. Same process is repeated for the buckling mode shapes of fixed column. Both graphs are shown in Figure 6.32.

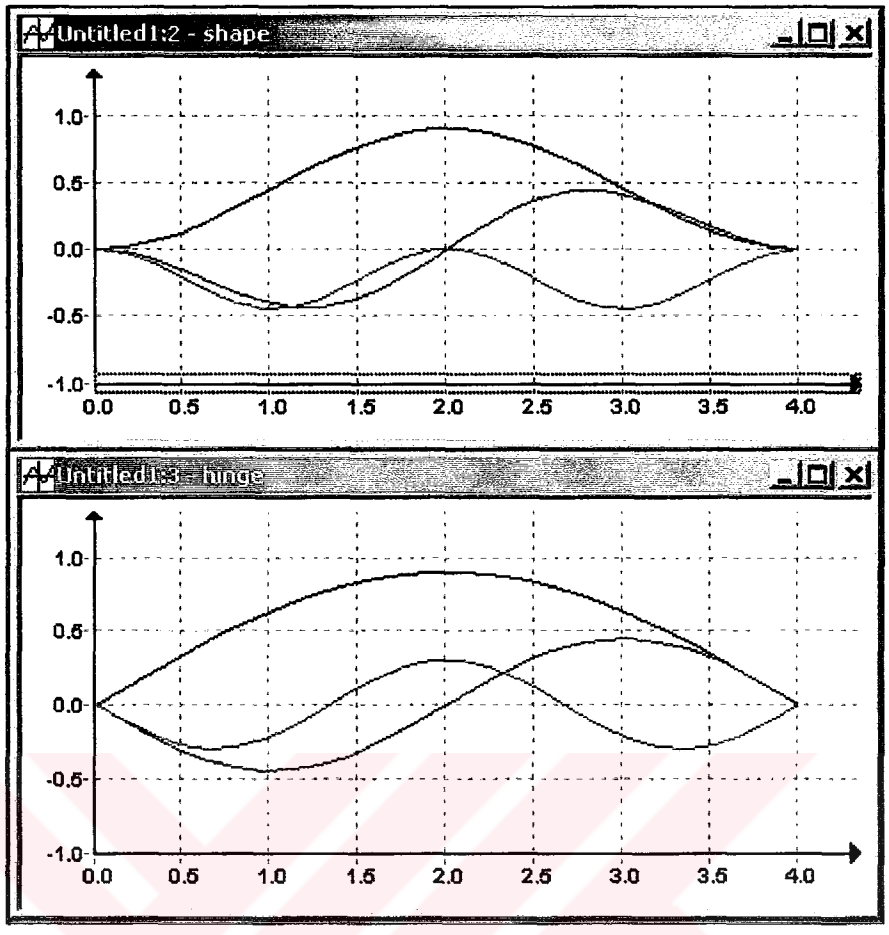


Figure 6.32 Buckling Mode Shapes for Fixed Column (Shown at the Top) and Hinged Column (Shown at the Bottom)

6.3.3 Results

As stated previously, only the first buckling load and mode shape have a practical meaning. Buckling loads for the hinged and fixed systems obtained from the buckling analysis are compared to the exact values given by the well-known critical Euler buckling load formula. Which is given by

$$P_{cr} = \frac{\pi^2 EI}{L^2} \text{ for the hinged column} \tag{6.6}$$

$$P_{cr} = \frac{4\pi^2 EI}{L^2} \text{ for the fixed column} \tag{6.7}$$

Results are summarized in Table 6.4.

Table 6.4 Critical Buckling Loads

Support Type	P_{cr} from Analysis (kN)	P_{cr} from Euler Formula (kN)
Fixed	321257.40	321255.62
Hinged	80313.93	80313.91

The slight differences in the results are due to the round-off errors in the Jacobi method.

6.4 Cantilever Beam Modeled With Plane Stress Elements

In this section, an example of cantilever beam modeled with plane stress elements is solved. This problem was taken from Reddy, 1993.

6.4.1 Problem Data

A cantilever beam is shown in Figure 6.33. The length of the beam is 10 in. Height and thickness of the beam are 2 in and 1 in, respectively. The beam is subjected to a uniformly distributed shear stress τ = 150 psi. The modulus of elasticity, $E = 30 \times 10^6$ psi and Poisson's ratio, $v = 0.25$.

Problem is solved for three different finite element meshes as shown in Figure 6.33.

The tip deflection is evaluated for each case. Results are summarized in Section 6.4.3

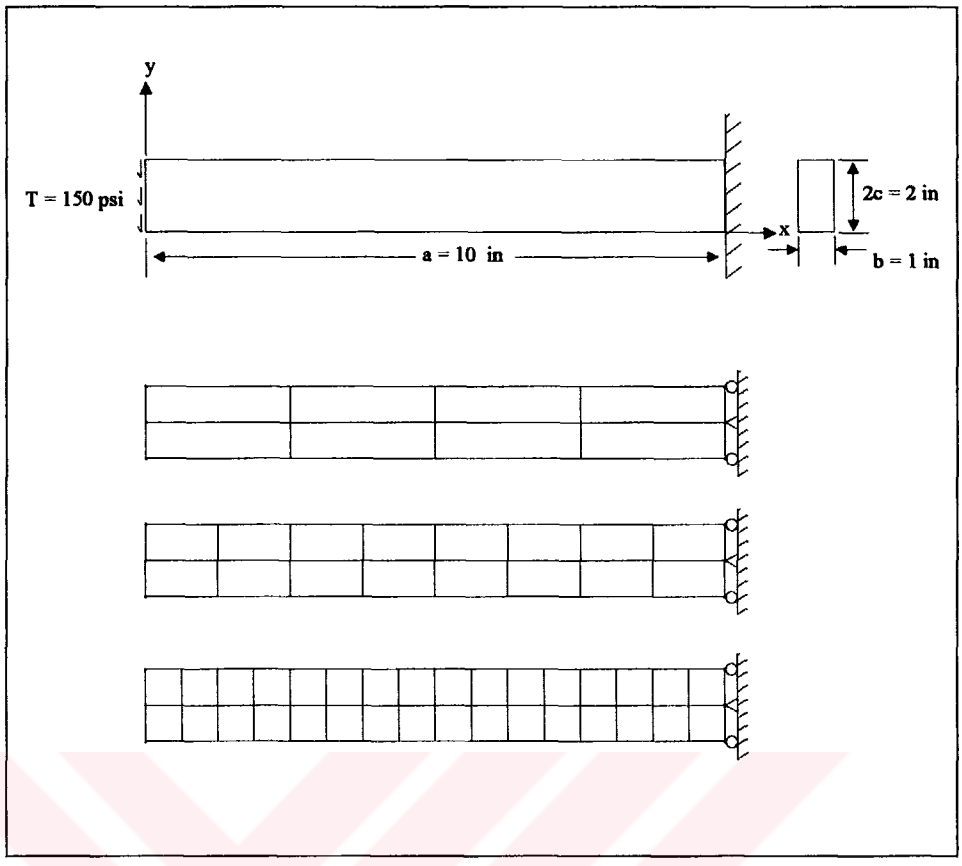


Figure 6.33 Cantilever Beam

6.4.2 Analysis in CALWin

Firstly the grid system is set as sown in Figure 6.34. Four spacing of 2.5 inches wide in X direction and two spacing of 1 inch wide in Y direction.

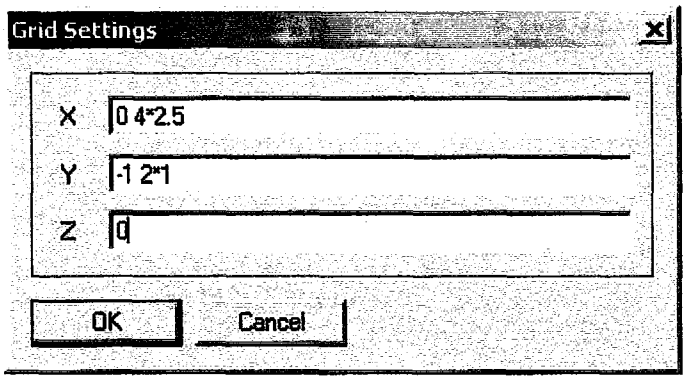


Figure 6.34 Grid Settings for 15-Node Mesh

Then, rectangular surface elements are defined using mouse. As in the case of creating line elements, mouse cursor will snap to nodal points and grid

intersection points as the surface element is drawn. The model is shown in Figure 6.35.

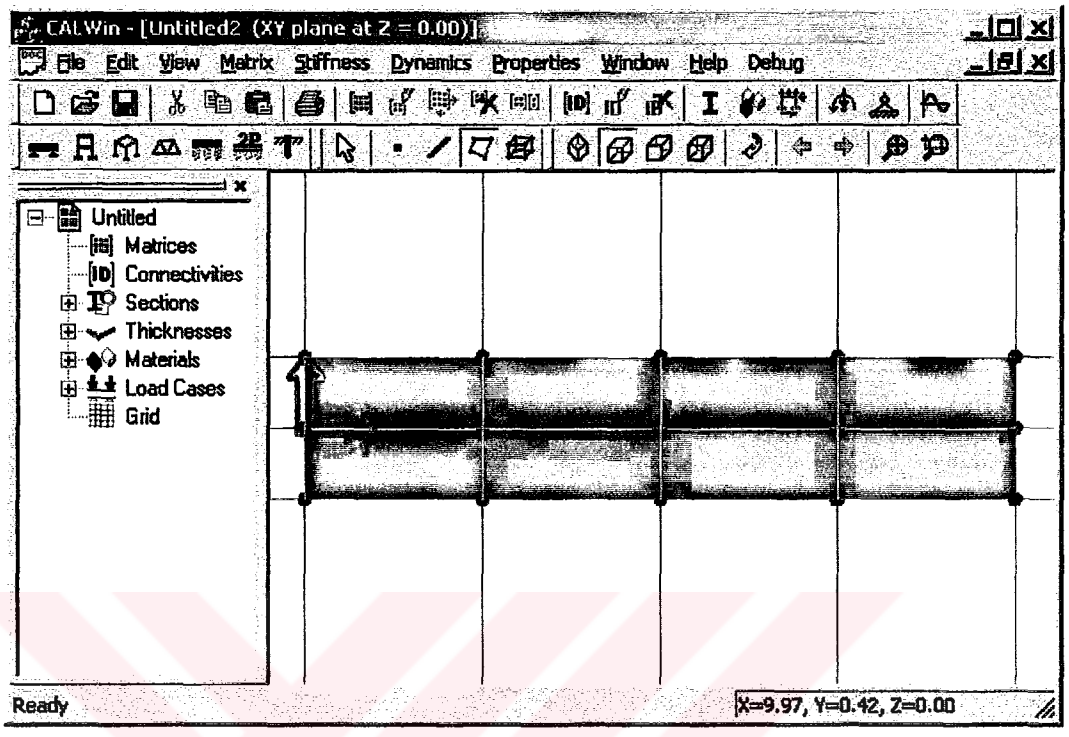


Figure 6.35 Surface Elements Defined

Unlike previous examples, this time degrees of freedom are numbered automatically by the program. However, this is a plane-stress problem and only translational degrees of freedom in XY plane are required. This is done in the *Options* dialog box, which is displayed by selecting *Options* command from the *File* menu.

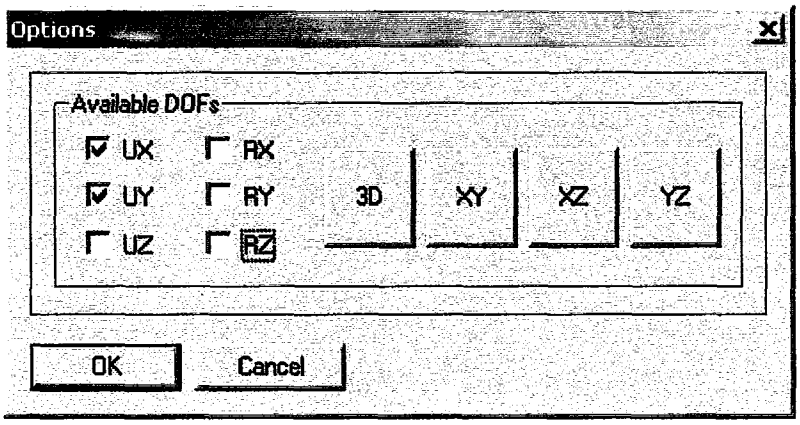


Figure 6.36 Options Dialog Box

Then, the support conditions are defined by assigning restraints to nodes. Restraints are assigned using *Node Property Editor,* which is shown in Figure 6.37.

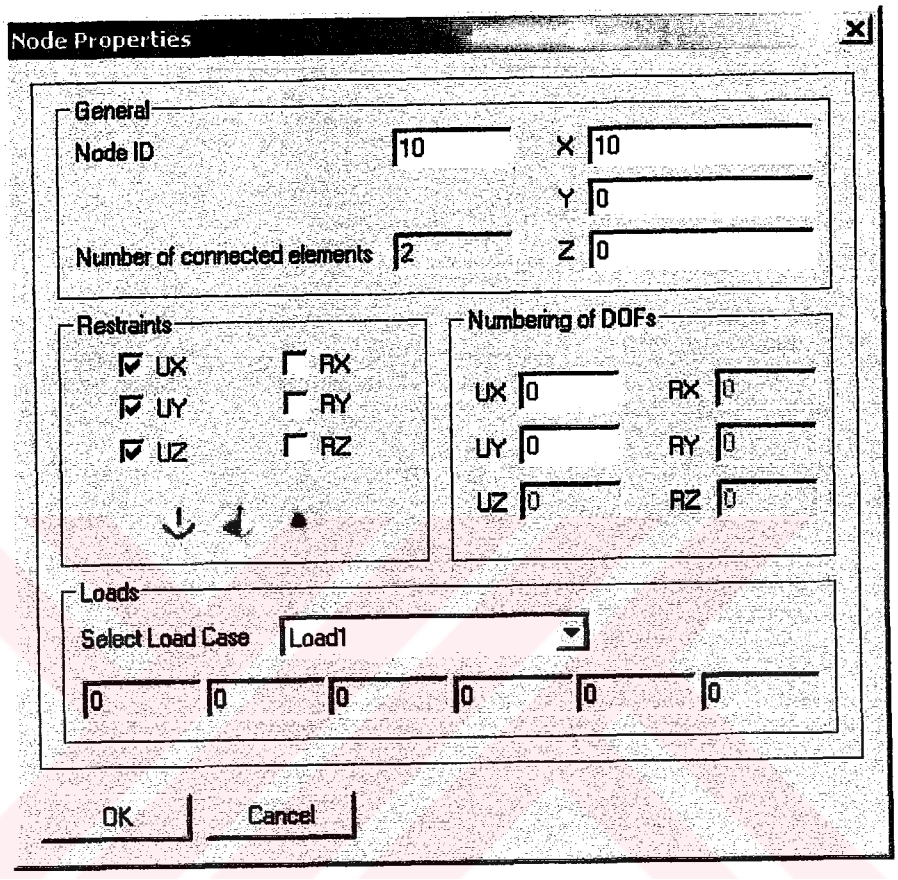


Figure 6.37 Node Property Editor

Next step is the definition of surface element properties. This includes the definition of thickness, analysis type, which is plane-stress analysis in this case and integration order for Gauss quadrature. These settings are done in *Surface Element Thickness Properties* dialog box as shown in Figure 6.38.

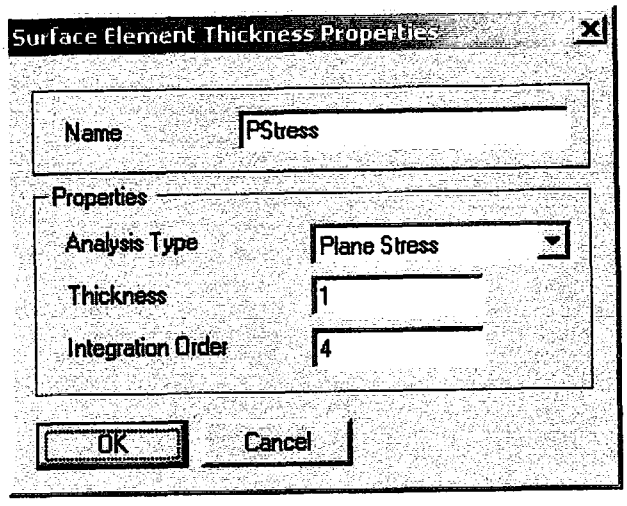


Figure 6.38 Surface Element Thickness Properties

Having defined the structure, loads are assigned to the nodes at the left end. In Figure 6.39, nodal loads of top and bottom nodes at the left end are shown. The load of the center node is 150 lb in negative Y-direction.

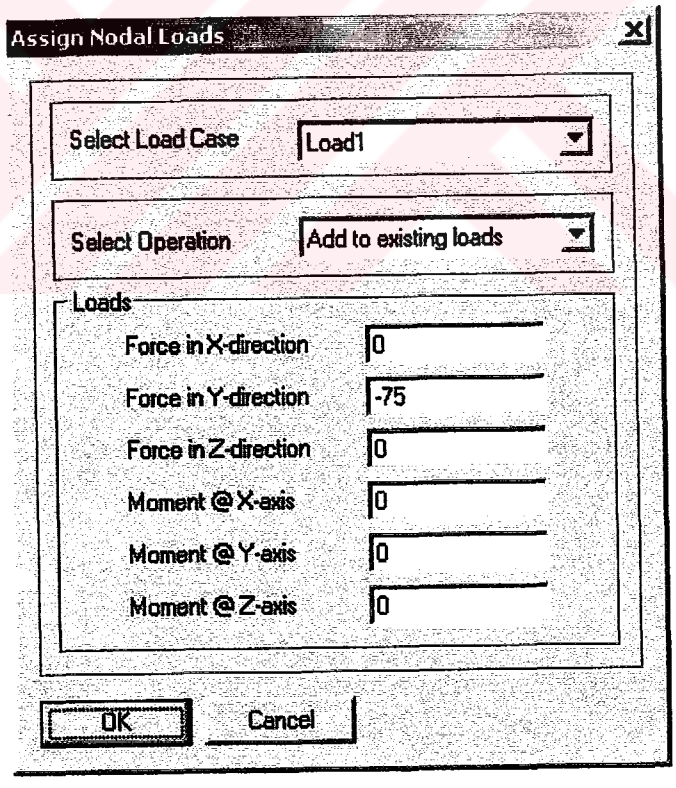


Figure 6.39 Nodal Load Assignment Dialog Box

Before, forming the structural stiffness matrix and load vector it is necessary to number the nodes which is accomplished by selecting the *Plane*

Numbering command under the menu item *Stiffness->Equation Numbering*. This command numbers the degrees of freedom considering available and restrained degrees of freedom.

Structural stiffness matrix and load vector can be formed automatically by CALWin using the *Assembly* dialog box which is displayed by running the *Assemble Element Matrices* command from the *Stiffness* menu item. This dialog is shown in Figure 6.40. Load vector is formed for the loads of selected load case.

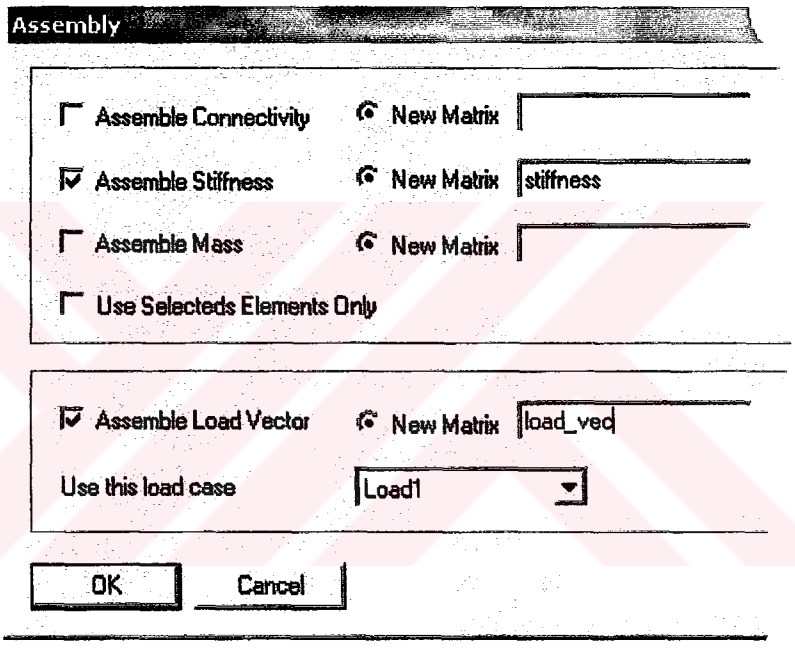


Figure 6.40 Element Matrix Assembly Dialog

Closing the dialog box with OK button, forms two matrices named *stiffness* and *load_vec* as shown in Figure 6.41. Analysis is completed by solving the equations. Equations are solved using the *Solve* command which is available in matrix operation dialog box.

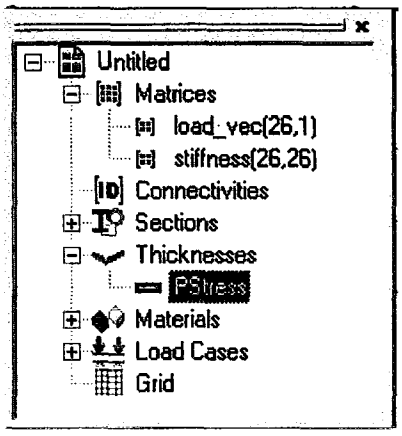


Figure 6.41 Structural Stiffness Matrix and Load Vector

6.4.3 Results

Results obtained from the analysis for three different finite element meshes are presented in Table 6.5 along with the results presented in Reddy (1993), and the elasticity solution

Table 6.5 Comparison of Finite Element Solution with Elasticity Solution

Number of Nodes	Tip Deflection (in)	
	CALWin	Reddy
15	-0.0031335	-0.0031335
27	-0.0043884	-0.0043884
51	-0.0048779	-0.0048779
Elasticity	0.51875	

6.5 Three-Dimensional Frame Structure

In this problem, static analysis of a two-story frame structure with unsymmetrical plan is performed. The results are compared with the analysis of the same structure using SAP2000.

6.5.1 Problem Data

Typical floor plan of a two-story steel frame structure is shown in Figure 6.42.

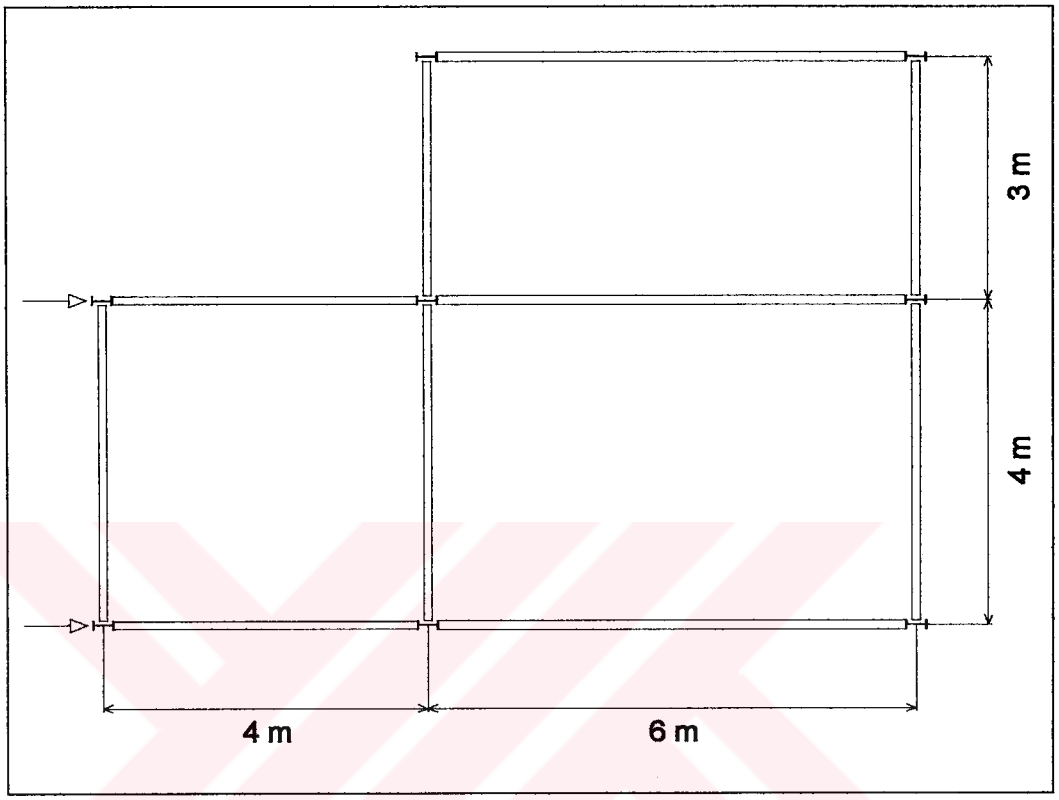


Figure 6.42 Typical Floor Plane of a Two-Story Steel Frame Structure

Building is subjected to lateral loads of 50kN at each story level applied tp leftmost to columns. All columns are IPE270. 6 meter-girders are IPE240 and the remaining ones are IPE200. Material is steel. Cross sectional properties are listed in Table 6.6 and material properties are shown in Table 6.7.

Table 6.6 Cross-Section Properties (In Units of Meter)

Section	Area	As2	As3	I33	I22	J
IPE200	2.850E-3	1.120E-3	1.417E-3	1.943E-5	1.420E-6	6.920E-8
IPE240	3.340E-3	1.298E-3	1.687E-3	2.772E-5	2.050E-6	9.030E-8
IPE270	4.590E-3	1.782E-3	2.295E-3	5.790E-5	4.200E-6	1.590E-7

Table 6.7 Material Properties (In Units of kN and m)

Unit Mass	Unit Weight	E	v	Alpha
7.827	76.8195	2E8	0.3	1.170E-5

6.5.2 Analysis in CALWin

As in the previous examples, first step is the setting of the grid system. Then the structure is defined using line elements and support conditions are defined by assigning restraints. After that, loading is defined by assigning nodal loads. These procedures have been presented in previous examples, thus they are not repeated here. The program looks like something in Figure 6.43 after the structure is defined.

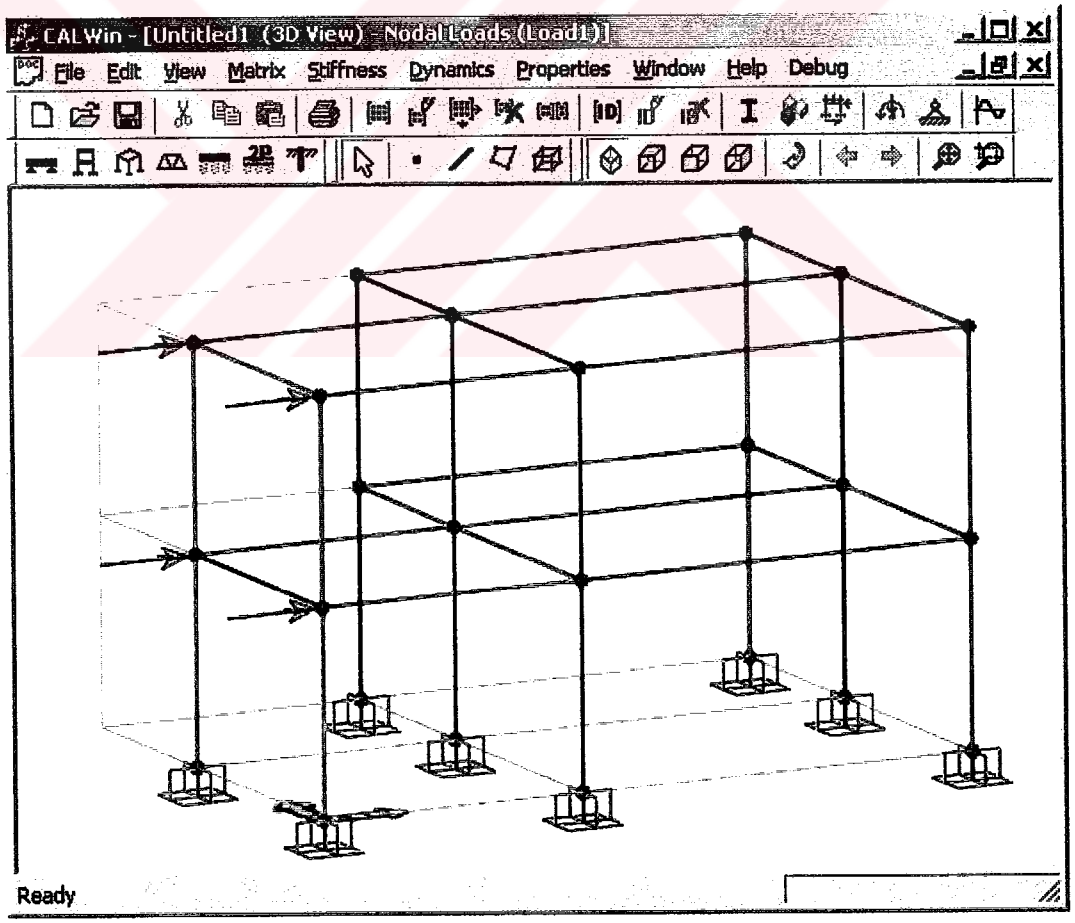


Figure 6.43 Three-Dimensional View of the Structure

Analysis is performed similar to the previous example. Firstly, equations are numbered by selecting the *Plane Numbering* command under the menu item *Stiffness->Equation Numbering*. Then the structural stiffness matrix and load vectors are formed as in Figure 6.40. Finally, equations are solved using the Solve command, which is available in matrix operation dialog box.

Next step is the evaluation of member forces. The element for which the member forces are to be evaluated is selected in the graphic editor (see Figure 6.43). Then the *Member Forces* command is executed from the *Stiffness* menu item. This will create a matrix containing the element forces in local coordinate system (see Figure 6.44).

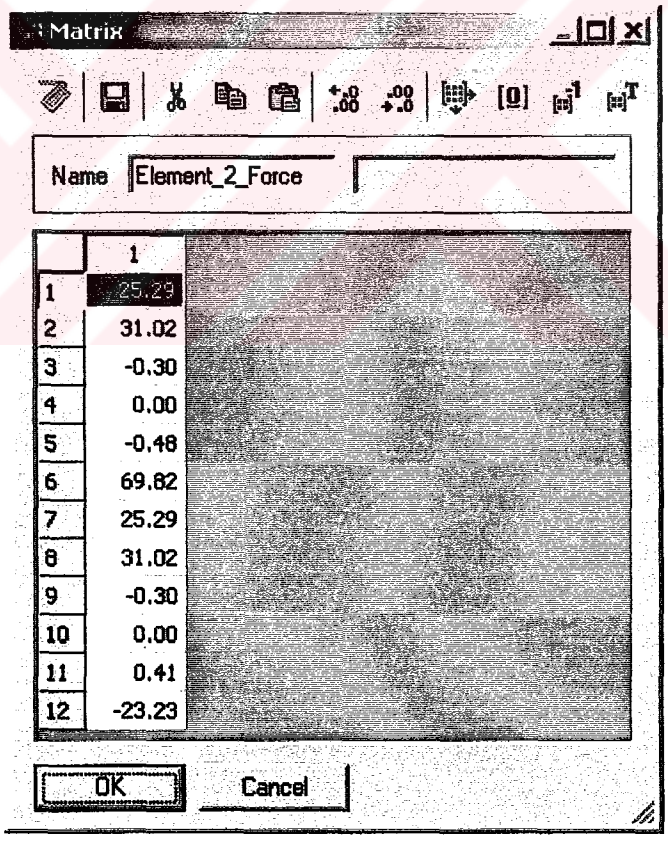


Figure 6.44 Element Forces of the Selected Element

6.5.3 Results

If the same structure is analyzed in SAP2000, a comparison of results shows that element forces and displacements are identical. For this reason, results of SAP2000 are not repeated here.

CHAPTER 7

SUMMARY AND CONCLUSIONS

7.1 Summary

With the exponential increase in the speed and capacity of computers, character based DOS programs was replaced in the last decade with programs that run on user-friendly graphical environment provided by Windows.

The primary objective of this study was to develop a computer program for educational purposes in structural analysis and dynamics that benefits from the graphical environment of Windows. The final outcome of this study, that is the software, was named as CALWin, since the working philosophy of the program and the computational algorithms used in the program was based on a former educational software called CAL91 (Wilson, 1991).

The second aim was to implement an isoparametric four-noded general quadrilateral and three-noded triangular plane stress/strain element, which was non-existent in finite element library of CAL91.

These objectives were achieved in four steps. Firstly, a set of C++ classes that handles the matrix and vector operations were developed. These matrix classes were utilized to perform all procedures of matrix structural analysis. In the next step, the coded algorithms in CAL91 were transformed to C++ code and

molded into objects. Third step was the creation of the graphical user interface. For this, the MFC (Microsoft Foundation Classes) library and OpenGL API (Application Programming Interface) was used. MFC provided the framework for the graphical user interface with serialization support, and OpenGL enabled the development of a 3D graphics engine almost as powerful as those of commercial software. Then, isoparametric elements were implemented into the program. Finally, a number of benchmark problems were solved in order to illustrate the running of CALWin.

7.2 Conclusions

Use of object oriented-programming approach resulted in a modular program structure and a more readable code. New finite elements can be added by deriving a class from the abstract base class and overriding the virtual functions.

The working principles of CAL91 were preserved while adding new automated features, such as analyzing the structure without dealing with any matrix operations or numbering the equations automatically. Therefore, program can be used for very simple tasks as simple as performing matrix operations, and for analysis of small sized structures.

Local coordinate systems of quadrilateral and triangular finite elements are set according to the viewing angle of the user. Thus, it is not necessary to define the nodes in a counterclockwise order.

7.3 Further Studies

Because of object-oriented programming paradigm used in the development of CALWin, it is fairly easy to extend the capabilities of the program

or to introduce new finite elements such as a shell element. Doing so is an obvious extension of the present study. Another further study may be the implementation of non-linear analysis procedures. Currently, CALWin does not have any non-linear analysis procedures. New equation numbering algorithms that minimize the profile of the stiffness matrix may be introduced into the program. This would significantly increase the capacity of the program.

REFERENCES

Alemdar, B. N., 1995, *An Exact Finite Element for Beams on Elastic Foundation*, M. Sc. Thesis, Middle East Technical University, Ankara.

Alemdar, B. N. and Gülkan, P., 1997, *Beams on Generalized Foundations: Supplementary Element Matrices*, Engineering Structures, Vol. 19, No. 11, 910-920.

Bathe, K. J. 1996, *Finite Element Procedures*, Prentice-Hall, New Jersey.

Blaszczak, M., 1999, *Professional MFC with Visual C++*, Wrox Press, Birmingham, UK, 155-236.

Chopra, A. K., 2001, *Dynamics of Structures: Theory and Application to Earthquake Engineering*, Prentice-Hall, New Jersey.

Eckel, B., 2000, *Thinking In C++*, Prentice-Hall, New Jersey, 23-27.

Horton, I., 1998, *Beginning Visual C++ 6*, Wrox Press, Birmingham, UK.

McGuire, W. and Gallagher, R. H., 2000, *Matrix Structural Analysis*, John Wiley & Sons.

Page-Jones, M., 2000, *Fundementals of Object-Oriented Design in UML*, Addison-Wesley, Massachusetts, 1-49.

Reddy, J. N., 1993, *An Introduction to the Finite Element Method* McGraw Hill, New Jersey, 456-468.

Smith, I. M. and Griffiths, D. V., 1998, *Programming the Finite Element Method*, John Wiley & Sons.

Stroustrup, B., 1997, *The C++ Programming Language*, Addison-Wesley, Massachusetts, 21-45

Wilson, E. L., 1998, *Three Dimensional Static and Dynamic Analysis of Structures*, Computers and Structures Inc., Berkeley, California.

Wilson, E.L., 1991, *CAL91: Computer Assisted Learning of Static and Dynamic Analysis of Structural Systems*, University of California, California.

Woo, M., Neider, J., Davis, T., Shreiner, D., 1999, *OpenGL Programming Guide*, -Addison-Wesley, Massachusetts, 27-86.

Yang, T. Y., 1986, *Finite Element Structural Analysis*, Prentice-Hall, New Jersey, 77-121.

APPENDIX

The source code of CALWin, executable files, some example problems solved with CALWin and copyright notice for the third party controls used in CALWin are provided in a compact disk that is attached to the inside of the back cover.

The source code is in a folder called "source" in the CD. Whole source code is in C++. Microsoft Visual Studio 6.0 or higher is required in order to compile the source code. However, it is also possible to compile under Borland C++ Builder 5 or higher provided that the necessary compiler settings are done.

The executable files can be found in the folder named "bin". These include the main EXE and required MFC DLLs.

An installation program is also provided to install CALWin. It is in the folder named "setup".

The examples solved in Chapter 6 can be accessed from the folder "examples". This folder contains the CALWin project files and some Excel files, which contain the charts and tables given in Chapter 6.