# ALADDIN : A COMPUTATIONAL TOOLKIT FOR INTERACTIVE ENGINEERING MATRIX AND FINITE ELEMENT ANALYSIS

**Mark Austin, Xiaoguang Chen
and Wane-Jang Lin**

Institute for Systems Research
and
Department of Civil Engineering
University of Maryland
College Park, MD 20742

November 17, 1995

## Abstract

This report describes Version 1.0 of ALADDIN, an interactive computational toolkit for the matrix and finite element analysis of engineering systems. The ALADDIN package is designed around a language specification, that includes quantities with physical units, branching constructs, and looping constructs. The basic language functionality is enhanced with external libraries of matrix and finite element functions.

ALADDIN's problem solving capabilities are demonstrated via the solution to a series of matrix and numerical analysis problems. ALADDIN is employed in the finite element analysis of two building structures, and two highway bridge structures.

# Contents

# Part I

# INTRODUCTION TO ALADDIN

# Chapter 1

# Introduction to ALADDIN

## 1.1 Problem Statement

This report describes the development and capabilities of ALADDIN (Version 1.0), an interactive computational toolkit for the matrix and finite element analysis of engineering systems. The current target application area for ALADDIN is design and analysis of traditional Civil Engineering structures, such as highway bridges and earthquake-resistant buildings. With literally hundreds of engineering analysis and optimization computer programs having been written in the past 10-20 years (see references [1, 6, 23, 24, 26, 25, 29, 30] for some examples), a reader might rightfully ask *who needs to write another engineering analysis package ?*

We respond to this challenge, and motivate the short- and long-term goals of this work by first noting that during the past two decades, computers have been providing approximately 25% more power per dollar per year. Advances in computer hardware and software have allowed for the exploration of many new ideas, and have been a key catalyst in what has led to the maturing of computing as a discipline. In the 1970's computers were viewed primarily as machines for research engineers and scientists – compared to today's standards, computer memory was very expensive, and central processing units were slow. Early versions of structural analysis and finite element computer programs, such as ABAQUS [1], ANSR [23], and FEAP [30] were written in the FORTRAN computer language, and were developed with the goal of optimizing numerical and/or instructional considerations alone. These programs offered a restricted, but well implemented, set of numerical procedures for static structural analyses, and linear/nonlinear time-history response calculations. And even though these early computer programs were not particularly easy to use, practising engineers gradually adopted them because they allowed for the analysis of new structural systems in a ways that were previously intractable.

During the past twenty years, the use of computers in engineering has matured to the point where importance is now placed on ease of use, and a wide-array of services being made available to the engineering profession as a whole. Computer programs written for engineering computations are expected to be fast and accurate, flexible, reliable, and of course, easy to use. Whereas an engineer in the 1970's might have been satisfied by a computer program that provided numerical solutions to a very specific engineering problem, the same engineer today might require the engineering analysis, plus computational support for design code checking, optimization, interactive computer graphics, network connectivity, and so forth. Many of the latter features are not a bottleneck for getting the job done. Rather, features such as interactive computer graphics simply make the job of describing a problem and interpreting results easier – the pathway from ease-of-use to productivity gains is well defined. It is also well worth noting that computers once viewed as a tool for computation alone, are now seen as an indispensable tool for computation and communications. In fact, the merging of computation and communications is making fundamental changes to the way an engineer conducts his/her day-to-day business activities. Consider, for example, an engineer who has access to a high speed personal computer with multimedia interfaces and global network connectivity, and who happens to be part of a geographically dispersed development team. The team members can use the Internet/E-mail for day-to-day communications, to conduct engineering analyses at remote sites, and to share design/analysis results among the team members. Clear communication of engineering information among the team members may be of paramount importance in determining the smooth development of a project.

The difficulty in following-up on the abovementioned hardware advances with appropriate software developments is clearly reflected in the economic costs of project development. In the early 1970's software consumed approximately 25% of total costs, and hardware 75% of total costs for development of data intensive systems. Nowadays, development and maintenance of software typically consumes more than 80% of the total project costs. This change in economics is the combined result of falling hardware costs, and increased software development budgets needed to implement systems that are much more complex than they used to be. Whereas one or two programmers might have written a complete program twenty years ago, today's problems are so complex that teams of programmers are needed to understand a problem and fill-in the details of required development.

When a computer program has a poorly designed architecture, its integration with another package can be very difficult, with the result often falling short of users' expectations. Let's suppose, for example, that we wanted to interface the finite element package FEAP [30] with the interactive optimization-based design environment called DELIGHT [6, 24, 29]. Since FEAP was not written with interfaces to external environments as a design criterion, a programmer(s) faced with this task would first need to figure out how FEAP and DELIGHT work (not an easy task), and then devise a mapping from DELIGHT's external interface routines to FEAP's subroutines. In the first writer's opinion, such a mapping is likely to exist, but only after several subroutines have been added to FEAP. The programmer(s) would need the computer skills and tenacity to stick-with the lengthy period of code development that would ensue. And what about the result ? In our experience [3, 4, 6], the integrated DELIGHT-FEAP tool would most likely do a very good job of solving a narrow range of problems, and as such, have a short life cycle. These barriers to integration are frustrating because finite element and optimization procedures are essentially specialized matrix computations – the disciplines should fit together in almost a seamless way. In our opinion, the main barrier to software integration is an ad-hoc approach to software tool development in the first place.

Rather than simply repeat the "scenario procedure" for yet another set of packages, this research project attempts to understand the structure matrix, finite element, and optimization packages should take so that they can be integrated in a natural way. Project ALADDIN begins with the design and implementation of a system specification that includes:

[1] `A Model` : The model will include data structures for the information to be stored, and a stack machine for the execution of the matrix and finite element algorithms.

[2] `A Language` : The language will be a means for describing the matrices and finite element mesh – it will act as the user interface to the underlying model.

    In traditional approaches to problem solving, engineers write the details of a problem and its solution procedure on paper. They use physical units to add clarity to the problem description, and may specify step-by-step details for a numerical solution to the problem. We would like the ALADDIN language to be textually descriptive, and strike a balance between simplicity and extensibility. It should use a small number of data types and control structures, incorporate physical units, and yet, be descriptive enough so that the *pencil and paper* and ALADDIN *problem description* files are almost the same.

[3] `Defined Steps and Ordering of the Steps` : The steps will define the transformations (e.g. nearly all engineering processes will require iteration and branching) that can be carried out on system components.

[4] `Guidance for Applying the Specification` : Guidance includes factors such as descriptive problem description files and documentation.

Our research direction is inspired in part by the systems integration methods developed for the European ESPRIT Project [20], and by the success of C. Although the C programming language has only 32 keywords, and a handful of control structures, its judicious combination with external libraries has resulted in the language being used in a very wide range of applications.

## 1.2 ALADDIN Components

Figure 1.1 is a schematic of the ALADDIN (Version 1.0) architecture, and shows its three main parts: (1) the kernel; (2) libraries of matrix and finite element functions, and (3) the input file(s).

Specific engineering problems are defined in ALADDIN problem description files, and solved using components of ALADDIN that are part interpreter-based, and part

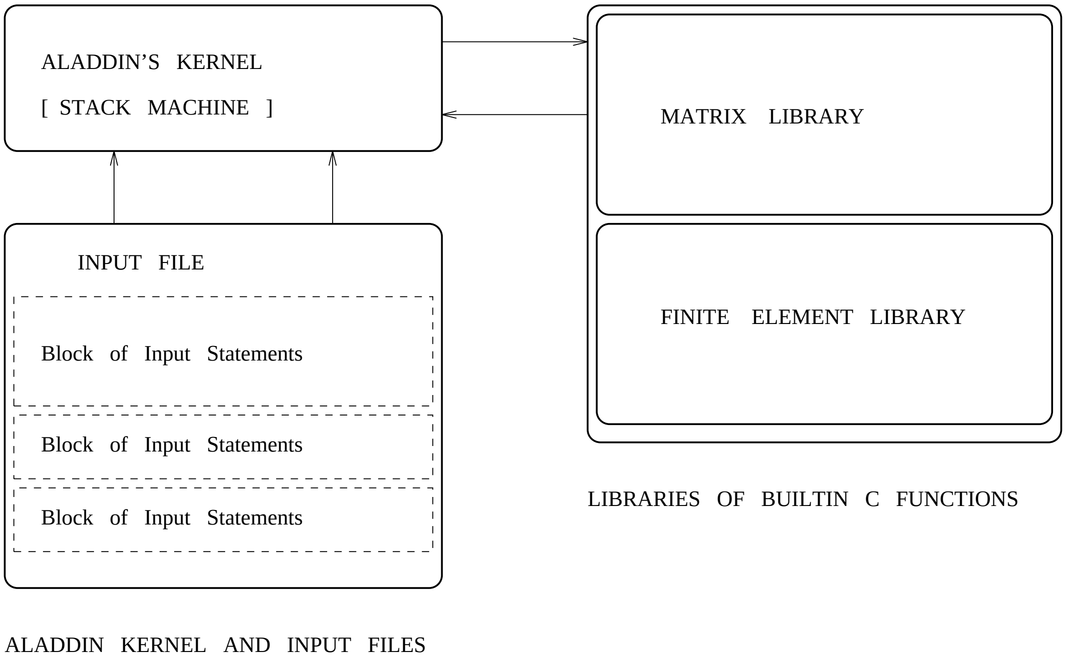


Figure 1.1: High Level Components in ALADDIN (Version 1.0)

compiled C code. It is important to keep in mind that as the speed of CPU processors increases, the time needed to prepare a problem description increases relative to the total time needed to work through an engineering analysis. Hence, clarity of an input file's contents is of paramount importance. In the design of the ALADDIN language we attempt to achieve these goals with: (1) liberal use of comment statements (as with the C programming language, comments are inserted between `/* ....  */`), (2) consistent use of function names and function arguments, (3) use of physical units in the problem description, and (4) consistent use of variables, matrices, and structures to control the flow of program logic.

ALADDIN problem descriptions and their solution algorithms are a composition of three elements: (1) data, (2) control, and (3) functions [27]:

[1] **Data :** ALADDIN supports three data types, "character string" for variable names, physical quantities, and matrices of physical quantities for engineering data. For example, the script of code

```
xCoord   = 2 m;
xVelociy = 2 m / sec;
```

defines two physical quantities, `xCoord` as 2 meters, and `xVelocity` as 2 meters per second. Floating point numbers are stored with double precision accuracy, and are viewed as physical quantities without units. There are no integer data types in ALADDIN.

[2] **Control :** Control is the basic mechanisms in a programming language for the specification of looping constructs and conditional branching.

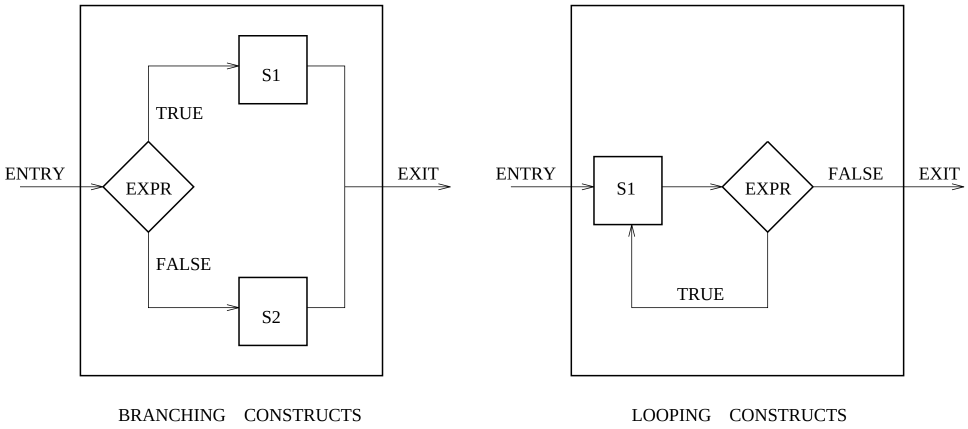


Figure 1.2: Branching and Looping Constructs in ALADDIN

In Chapter 2 we will see that ALADDIN supports the `while` and `for` looping constructs, and the `if` and `if-then-else` branching constructs. The data and control components of the ALADDIN language are implemented as a finite-state stack machine model, which follows in the spirit of work presented by Kernighan and Pike [18]. ALADDIN's stack machine reads blocks of command statements from either a problem description file, or the keyboard, and converts them into an array of machine instructions. The stack machine then executes the statements.

[**3**] **Functions :** The functional components of ALADDIN provide hierarchy to the solution of our matrix and finite element processes, and are located in libraries of compiled C code, as shown on the right-hand side of Figure 1.1. Version 1.0 of ALADDIN has functional support for basic matrix operations, that includes numerical solutions to linear equations, and the symmetric eigenvalue problem. A library of finite element functions is provided.

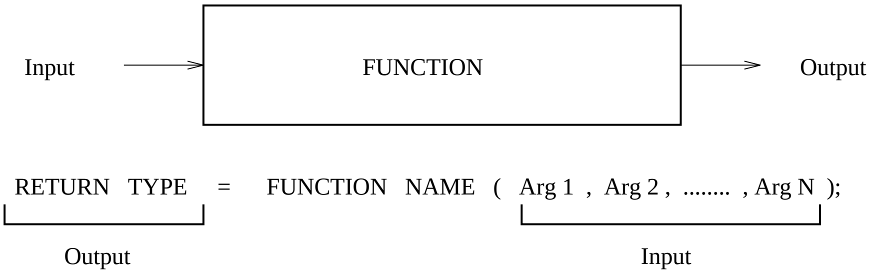


Figure 1.3: Schematic of Functions in ALADDIN

Figure 1.3 shows the general components of a function call, it's input argument list, and the function return type. A key objective in the language design is to devise families of library functions that will make ALADDIN's problem solving ability as wide as possible.

Version 1.0 of ALADDIN does not support user-defined functions in the input-file (or keyboard) command language.

The strategy we have followed in ALADDIN's development is to keep the number and type of arguments employed in library function calls small. Whenever possible, the function's return type and arguments should be of the same data type, thereby allowing the output from one or more functions to act as the input to following function call. More precisely, we would like to write input code that takes the form

```
returntype1 = Function1();                              /* <= application area 1 */
returntype2 = Function2();                              /* <= application area 1 */

returntype3 = Function3( returntype1, returntype2 );  /* <= application area 2 */
```

or even

```
returntype3 = Function3( Function1(), Function2() );
```

If `Function1()` and `Function2()` belong to application area 1 (e.g. matrix analysis), and `Function3()` belongs to application area 2 (e.g. finite element analysis; optimization), then this language structure allows application areas 1 and 2 to be combined in a natural way. In Chapters 2 to 6, we will see that most of the built-in functions accept one or two matrix arguments, and return one matrix argument. Occasionally, we will see built-in function that have more than two function arguments, and return a quantity instead of a matrix. Collections of quantities are returned from functions by bundling them into a single matrix – the individual quantities are then extracted as elements of the function return type.

| Factor | Interpreted Code | Compiled Code |
|---|---|---|
| User Control | High | Very Low |
| Required User Knowledge | High | Medium |
| Speed of Execution | Slow | Fast |

Table 1.1: Trade-Offs – Interpreted Code Versus Compiled Code

Table 1.1 contains a summary of trade-offs between the use of interpreted code versus compiled C code. A key advantage of interpreters is flexibility – problem parameters and algorithms may be modified during the problem solving process, and without having to recompile source code. This feature reduces the time needed to work through an interation of the problem solving process. The well known trade-off is speed of execution. Interpreted code executes much slower – approximately ten times slower – than compiled C code. Consequently, we expect that as ALADDIN evolves, new algorithms will be developed in tested as interpreted code, and once working, will be converted into libraries of compiled C code having an interface that fits-in with the remaining library functions.

## 1.3 Scope of this Report

This report is divided into four parts. In Part II we will see that the ALADDIN language supports variable arithmetic with physical units, matrix operations with units, and looping and branching control structures, where decisions are made with quantities having physical units. Chapters 3 and 4 demonstrate use of the ALADDIN language via a suite of problems – we compute the roots of nonlinear equations, demonstrate the Han-Powell method of optimization, and solve several problems from structural engineering and structural dynamics.

Part III of this report describes the built-in functions for the generation of finite element meshes, external loads, and finite element modeling assumptions. We demonstrate these features by solving a variety two- and three-dimensional finite element problems.

We have written Part IV for ALADDIN developers, who may need to understand the data structures and algorithms used to read and store matrices, to create and store finite element meshes, and to construct the stack machine. Readers of this section are assumed to have a good working knowledge of the C programming language.

# Part II

# MATRIX LIBRARY

# Chapter 2

# Command Language for Quantity and Matrix Operations

The purposes of this chapter are to explain execution of the ALADDIN environment, and details of the matrix command language within ALADDIN. For ease of reading, and when space permits, output from ALADDIN is juxtaposed with the input command(s).

## 2.1 How to Start (and Stop) ALADDIN

ALADDIN's command line arguments are setup in a very flexible way. The first argument is simply `ALADDIN`, the name of the executable program.

**Input from Keyboard**

```
ALADDIN -[ks]
```

**Input from a File**

```
ALADDIN -[sf] <filename>
```

Command options have the following meaning:

`-f` Indicate input from a file (it must be accompanied by a filename).

`-k` Indicates input from the keyboard. When input is generated via the keyboard, a input history file will be generated and named as **inputfile.std**. In the likely event of mistyping or a syntax error, you can continue type in **inputfile.std**.

`-s` All input will be scanned for syntax errors without actually executing the program.

The command `ALADDIN` must be accompanied by either the `-f` flag, or the `-k` flag. The `-s` flag is optional. Command line options can be typed in arbitrary sequences.

The command to exit from ALADDIN is

```
quit;
```

Don't forget the semicolon.

## 2.2 Format of General Command Language

The ALADDIN command language corresponds to individual statements, and sequences of statements.

```
statement 1;
statement 2;
........
statement N;
```

Each statement ends with a semi-colon (;). Comment statements (as with the C programming language, comments are enclosed between `/* ....  */`), as in

```
/*
 *  ================================
 *  Here is a block of N statements.
 *  ================================
 */

statement 1;         /* the  first ALADDIN statement */
statement 2;         /* the second ALADDIN statement */
........
statement N;         /* the   n^th ALADDIN statement */
```

Basic output of character strings and physical quantities is handled with the `print` command. For example,

```
print "Here is one line of output \n";
```

gives

```
       Here is one line of output
```

Character strings are enclosed within quotes (i.e `"....."`). The escape character ($\backslash n$) forces output onto a new line.

## 2.3 Physical Quantities

While the importance of the engineering units is well known [11, 13, 17], physical units are not a standard part of many main-stream finite element software packages – indeed, most engineering software packages simply hold the engineer responsible for making sure engineering units are consistent, While this practice of implementation may be satisfactory for computation of well established algorithms, it is almost certain to lead to incorrect results when engineers are working on the development of new and innovative computations. ALADDIN deviates from this trend by providing support for basic arithmetic operations on quantities and matrices of physical quantities.

### 2.3.1 Definition and Printing of Quantities

Let's begin with the basics. A quantity is a number with physical units. To assign the quantity "2 m" to variable "x" just type

```
x = 2 m;
```

The semi-colon character ";" is required for every command to indicate the end of the one statement. The following script of code demonstrates definition of quantities.

INPUT

```
print " LENGTH UNITS : SI SYSTEM \n";

x =  1 mm; y =  1 cm; z =  1 dm;
u =  1  m; v =  1 km;

print  "x = ", x, "\n";
print  "y = ", y, "\n";
print  "z = ", z, "\n";
print  "u = ", u, "\n";
print  "v = ", v, "\n";

print "\n VOLUME UNITS : US \n";

x =  1 gallon; y =  1 barrel;

print  "x = ", x, "\n";
print  "y = ", y, "\n";

print "\n MASS UNITS : SI SYSTEM \n";

x = 1 g; y = 1 kg; z = 1 Mg;

print  "x = ", x, "\n";
print  "y = ", y, "\n";
print  "z = ", z, "\n";
```

OUTPUT

```
LENGTH UNITS : SI SYSTEM

x =        1 mm
y =        1 cm
z =        1 dm
u =        1 m
v =        1 km

VOLUME UNITS : US

x =        1 gallon
y =        1 barrel

MASS UNITS : SI SYSTEM

x =        1 g
y =        1 kg
z =        1 Mg
```

```
print " TIME UNITS : SI SYSTEM \n";

x =  1 sec; y =  1  ms;
z =  1 min; u =  1  hr;

print  "x = ", x, "\n";
print  "y = ", y, "\n";
print  "z = ", z, "\n";
print  "u = ", u, "\n";
```

```
TIME UNITS : SI SYSTEM

x =        1 sec
y =        1 ms
z =        1 min
u =        1 hr
```

```
print "\n TEMPERATURE UNITS : SI SYSTEM \n";
x = 1 deg_C;

print  "x = ", x, "\n";
```

```
TEMPERATURE UNITS : SI SYSTEM

x =        1 deg_C
```

```
print "\n TEMPERATURE UNITS : US SYSTEM \n";
x = 1 deg_F;

print  "x = ", x, "\n";
```

```
TEMPERATURE UNITS : US SYSTEM

x =        1 deg_F
```

```
print "\n UNITS OF FREQUENCY & SPEED \n";
x = 1 Hz;
y = 1 rpm;     /* rev. per min  */
z = 1 cps;     /* cycle per sec */

print  "x = ", x, "\n";
print  "y = ", y, "\n";
print  "z = ", z, "\n";
```

```
UNITS OF FREQUENCY & SPEED

x =        1 Hz
y =        1 rpm
z =        1 cps
```

```
print "\n FORCE UNITS : SI SYSTEM \n";

x =  1   N; y =  1  kN; z =  1 kgf;

print  "x = ", x, "\n";
print  "y = ", y, "\n";
print  "z = ", z, "\n";
```

```
FORCE UNITS : SI SYSTEM

x =        1 N
y =        1 kN
z =        1 kgf
```

```
print "\n PRESSURE UNITS : SI SYSTEM \n";

x =  1   Pa; y =  1  kPa;
z =  1  MPa; u =  1  GPa;

print  "x = ", x, "\n";
print  "y = ", y, "\n";
print  "z = ", z, "\n";
print  "u = ", u, "\n";
```

```
PRESSURE UNITS : SI SYSTEM

x =        1 Pa
y =        1 kPa
z =        1 MPa
u =        1 GPa
```

```
print "\n ENERGY UNITS : SI SYSTEM \n";

x =  1 Jou; y =  1  kJ;

print  "x = ", x, "\n";
print  "y = ", y, "\n";
```

```
ENERGY UNITS : SI SYSTEM

x =        1 Jou
y =        1 kJ
```

```
print "\n POWER UNITS : SI \n";                          POWER UNITS : SI

x =  1 Watt; y =  1  kW;

print  "x = ", x, "\n";                                   x =       1 Watt
print  "y = ", y, "\n";                                   y =       1 kW

print "\n UNITS OF PLANE ANGLE UNITS \n";                 UNITS OF PLANE ANGLE UNITS

x =  1 deg; y =  1 rad;

print  "x = ", x, "\n";                                   x =       1 deg
print  "y = ", y, "\n";                                   y =       1 rad
```

These examples focus on the SI system of units. In the US system, units of length are provided for micro inches (`micro_in`), inches (`in`), feet (`ft`), yards (`yard`), and miles (`mile`). Similarly, US units of mass are provided for pounds (`lb`), grains (`grain`), kilo pounds (`klb`), and tons (`ton`). Units of force in the US system are pounds force (`lbf`) and one thousand pounds force (`kips`). Corresponding units of pressure in the US system are pounds per square inch (`psi`), and thousands of pounds per square inch (`ksi`).

**Restrictions on Quantity Names :** Like many other programming languages, the ALADDIN command language has keywords and constants which are reserved for special purposes – they cannot be used for arbitrary purposes, such as variable names. For example, ALADDIN reserves the characters "N" and "m" for engineering units Newton and metre. Similar restrictions apply to all units, keywords defining looping and control constructs (e.g. `for` and `while`), as well as built-in function names.

A complete list of keywords and constants is given in Table 2.1. Table 2.2 contains a list of names that are reserved for mathematical functions. Reserved names for matrix allocation functions, and functions to compute matrix operations, are given in Tables 2.3 and 2.4, respectively. Reserved names also apply to names of functions for solving linear equations, eigenvalues and eigenvectors, as well as finite element analysis. These names are listed in Tables 2.5, 2.6, 5.1, 5.2, and in the text of Chapter 5.

### 2.3.2 Formatting of Quantity Output

The command option `"(< units >)"` allows for the printing of a quantity with a desired units. The following script of code gives some examples:

```
----------------INPUT----------------      ----------------OUTPUT----------------

print " LENGTH UNITS \n";                              LENGTH UNITS

v =  1 km; w =  1 mile;
```

```
print  "v    = ", v       , "\n";                    v    =        1 km
print  "v    = ", v (in)  , "\n";                    v    = 3.937e+04 in
print  "v    = ", v (ft)  , "\n";                    v    =     3281 ft
print  "v    = ", v (yard), "\n";                    v    =     1094 yard
print  "v    = ", v (mile), "\n";                    v    =   0.6214 mile

print  "w    = ", w       , "\n";                    w    =        1 mile
print  "w    = ", w (mm)  , "\n";                    w    = 1.609e+06 mm
print  "w    = ", w (cm)  , "\n";                    w    = 1.609e+05 cm
print  "w    = ", w (dm)  , "\n";                    w    = 1.609e+04 dm
print  "w    = ", w (m )  , "\n";                    w    =     1609 m
print  "w    = ", w (km)  , "\n";                    w    =    1.609 km

print "\n VOLUME UNITS \n";                          VOLUME UNITS
x =  1 gallon;

print  "x = ", x     ,     "\n";                     x =        1 gallon
print  "x = ", x (m^3),    "\n";                     x = 0.003785 m^ 3.0
print  "x = ", x (ft^3),   "\n";                     x =   0.1337 ft^ 3.0
print  "x = ", x (cm^3),   "\n";                     x =     3785 cm^ 3.0
print  "x = ", x (in^3),   "\n";                     x =      231 in^ 3.0
print  "x = ", x (barrel), "\n";                     x =  0.02381 barrel

print "\n TEMPERATURE UNITS \n";i                    TEMPERATURE UNITS

x = 1 deg_C;    U = 10 mm/DEG_C;
y = 1 deg_F;    V = 10 in/DEG_F;

print  "x = ", x        , "\n";                      x =        1 deg_C
print  "x = ", x (deg_F), "\n";                      x =     33.8 deg_F
print  "y = ", y        , "\n";                      y =        1 deg_F
print  "y = ", y (deg_C), "\n";                      y =   -17.22 deg_C
print  "U = ", U, "\n";                              U =       10 mm/DEG_C
print  "V = ", V, "\n";                              V =       10 in/DEG_F

print "\n";
print "U*(1 DEG_C) = ", U*(1 DEG_C) (mm), "\n";     U*(1 DEG_C) =       10 mm
print "U*(1 DEG_F) = ", U*(1 DEG_F) (mm), "\n";     U*(1 DEG_F) =    5.556 mm

print "V*(1 DEG_C) = ", V*(1 DEG_C) (in), "\n";     V*(1 DEG_C) =       18 in
print "V*(1 DEG_F) = ", V*(1 DEG_F) (in), "\n";     V*(1 DEG_F) =       10 in

print " TIME UNITS \n";                              TIME UNITS
x =  1  hr;

print  "x = ", x, "\n";                              x =        1 hr
print  "x = ", x (min), "\n";                        x =       60 min
print  "x = ", x (sec), "\n";                        x =     3600 sec

print "\n UNITS OF FREQUENCY & SPEED \n";            UNITS OF FREQUENCY & SPEED

y = 60 rpm;     /* rev. per min  */

print  "y = ", y,       "\n";                        y =       60 rpm
```

```
print  "y = ", y (Hz),  "\n";                      y =        1 Hz
print  "y = ", y (cps), "\n";                      y =        1 cps

print "\n PLANE ANGLE UNITS \n";                   PLANE ANGLE UNITS

x =  180 deg; y =  PI;

print  "x = ", x, "\n";                            x =      180 deg
print  "x = ", x (rad), "\n";                      x =    3.142 rad
print  "y = ", y, "\n";                            y =   3.1416e+00
print  "y = ", y (deg), "\n";                      y =      180 deg
```

Similar conversion factors exist for units of mass, force, pressure, energy and power.

### 2.3.3 Quantity Arithmetic

Physical units may be manipulated with basic multiply, division and power operations. The following script of code demonstrates the range of arithmetic operations that are possible:

```
------------- INPUT -------------               ------------- OUTPUT -------------

x  = 10 g; y  = 1 kg; z  = 1  m;

print "\nADDITION & SUBTRACTION \n";                ADDITION & SUBTRACTION

print  "x + y = ", x + y, "\n";                     x + y =     1.01 kg
print  "x - y = ", x - y, "\n";                     x - y =    -0.99 kg

print "\n MULTIPLY \n";                             MULTIPLY
print  "x * y = ", x * y, "\n";                     x * y =       10 g.kg
print  "x * z = ", x * z, "\n";                     x * z =       10 g.m

print "\n DIVISION \n";                             DIVISION
print  "x / y   = ", x / y, "\n";                   x / y   =     1.0000e-02
print  "x / z   = ", x / z, "\n";                   x / z   =       10 g/m
print  "1 / z   = ", 1 / z, "\n";                   1 / z   =        1 m^-1.0
print  "z / 2   = ", z / 2 ,"\n";                   z / 2   =      0.5 m
print  "1/(x*y) = ", 1/(x*y),"\n";                  1/(x*y) =      0.1 1/(g.kg)
print  "z/(x*y) = ", z/(x*y),"\n";                  z/(x*y) =      0.1 m/g.kg
print  "(x*y)/z = ", (x*y)/z,"\n";                  (x*y)/z =       10 g.kg/m

print "\n MATH CALCULATIONS \n";                    MATH CALCULATIONS
u  = 30 deg;
v  =  -PI/2;
w  =    2;
print  "sin(u)     = ", sin(u),     "\n";           sin(u)     =     5.0000e-01
print  "sin(v)     = ", sin(v),     "\n";           sin(v)     =    -1.0000e+00
print  "cos(u)     = ", cos(u),     "\n";           cos(u)     =     8.6603e-01
print  "cos(v)     = ", cos(v),     "\n";           cos(v)     =     6.1230e-17
```

```
print  "tan(PI/4)  = ", tan(PI/4),  "\n";          tan(PI/4)  =     1.0000e+00
print  "atan(1)    = ", atan(1),    "\n";          atan(1)    =     7.8540e-01
print  "abs(-3 m)  = ", abs(-3 m),  "\n";          abs(-3 m)  =        3 m
print  "abs(v)     = ", abs(v),     "\n";          abs(v)     =     1.5708e+00
print  "log(1E+5)  = ", log(1E+5),  "\n";          log(1E+5)  =     1.1513e+01
print  "log10(1E+5)= ", log10(1E+5),"\n";          log10(1E+5)=     5.0000e+00
print  "exp(w)     = ", exp(w),     "\n";          exp(w)     =     7.3891e+00
```

We use the print command and the character string "\n" containing the newline to print physical quantities and statements. Characters inside the equation marks " " are considered statements. Arguments to mathematical functions, such as log() and exp() take dimensionless quantities. Trigonometric functions can, however, take arguments with the units degree and radian because the latter are non-dimensional.

Power operations on quantities do not work in quite the same way as the basic add, subtract, multiply and divide operations on quantities. If quantity1 and quantity2 are physical quantities, then the operation

```
quantity1 ^ quantity2;
```

is defined for dimensionless quantity1 and quantity2, and cases where one of the two operands, but not both operands, has units. Unlike the basic arithmetic operations, which are evaluated left-to-right, expressions involving power operations are evaluated from right-to-left. Here are some examples:

```
--------------INPUT--------------          -------------OUTPUT-------------

print "\n POWER \n";                                POWER

print "2^2^3       = ", 2^2^3         , "\n";       2^2^3      =        256
print "(2 m)^2     = ", (2 m)^2       , "\n";       (2 m)^2    =          4 m^2
print "2^2 m       = ", 2^2 m         , "\n";       2^2 m      =          4 m
print "2^2^2 m     = ", 2^2^2 m       , "\n";       2^2^2 m    =         16 m
print "10^4   N/m = ", 10^4    N/m  , "\n";        10^4   N/m =      1e+04 N/m
print "10^2^2 N/m = ", 10^2^2  N/m  , "\n";        10^2^2 N/m =      1e+04 N/m

print "0.25*2^2   m  = ", 0.25*2^2   m, "\n";      0.25*2^2 m =          1 m
print "1/2*2      m  = ",  1/2*2      m, "\n";     1/2*2    m =          1 m
print "(1/2*2     m) = ", (1/2*2     m), "\n";     (1/2*2  m) =          1 m
print "(1/2^2     m) = ", (1/2^2     m), "\n";     (1/2^2  m) =     0.25 1/m
print "(1/2^2)*(1 m) = ", (1/2^2)*(1 m), "\n";     (1/2^2)*(1 m) =  0.25 m
print "1/2^2*(1 m)   = ",  1/2^2 *(1 m), "\n";      1/2^2*(1 m)  =  0.25 m

x = 100 kg; z = 10 m;

print "sqrt(100 kg)  = ", sqrt(x) , "\n";          sqrt(100 kg)  =  10 kg^0.5
print "z^2           = ", z^2 , "\n";              z^2           =  100 m^2
print "z^-3          = ", z^-3, "\n";              z^-3          =  0.001 1/m^3
print "(x*z)^1       = ", (x*z)^1, "\n";           (x*z)^1       =  1000 m.kg
print "(x*z)^2       = ", (x*z)^2, "\n";           (x*z)^2       =  1e+06 m^2.kg^2
```

As mentioned above, power operations such as

```
x = (1 m)^(2 sec);
```

are illegal, and will cause ALADDIN to terminate its execution.

An important feature of quantity operations is the check for consistent units. Suppose, for example, we try to add a quantity with time units to a second quantity having units of length, ALADDIN provides an appropriate fatal error message

```
x = 1 in; y = 1 sec;

z = x + y;
FATAL ERROR >> In Add() : Inconsistent Dimensions.
FATAL ERROR >> Compilation Aborted.
```

followed by the termination of program execution.

### 2.3.4 Making a Quantity Dimensionless

In the development of algorithms to solve engineering problems, we will sometimes need to strip (or remove) the units from a physical quantity, and work with the numerical value alone. The built-in function `QDimenLess()` removes units from quantities, as demonstrated in the following script of code:

INPUT

```
print "\n MAKE A QUANTITY DIMENSIONLESS \n";

x = 1 N; y = 1 cm/sec;

z = QDimenLess(x); u = QDimenLess(y);

print  "x (with dimen)    = ", x, "\n";
print  "y (with dimen)    = ", y, "\n";
print  "x (without dimen) = ", z, "\n";
print  "y (without dimen) = ", u, "\n";
```

OUTPUT

```
MAKE A QUANTITY DIMENSIONLESS

x (with dimen)    =          1 N
y (with dimen)    =       0.01 m/sec
x (without dimen) =   1.0000e+00
y (without dimen) =   1.0000e-02
```

### 2.3.5 Setting Units Type to US or SI

The default units type is `SI`. The function `SetUnitsType("string")`, where "string" equals `"SI"` or `"US"`, sets the units to SI or US, respectively.

You should call `SetUnitsType()` before the finite element displacements and stresses are printed (i.e. using `PrintDispl()` and `PrintStress()`).

### 2.3.6 Switching Units On and Off

The command `SetUnitsOn` turns the checking of units on, and the command `SetUnitsOff` turns the checking of units off. The default units option is `SetUnitsOn`.

| Category | List of Reserved Names |
|---|---|
| Constants | DEG = 57.29577951308, E = 2.718281828459, GAMMA = 0.577215664901, and PI = 3.141592653589 |
| Keywords | break, else, for, if, print, quit, read, return, SetUnitsOn, SetUnitsOff, then, while |
| SI Units | micron, mm, cm, dm, m, km, g, kg, mg, N, kN, kgf, Pa, kPa, MPa, GPa, deg_C, DEG_C, Jou, kJ, Watt, kW, Hz, rpm, cps |
| US Units | in, ft, yard, mile, mil, micro_in, lb, klb, ton, grain, lbf, kips, psi, ksi, deg_F, DEG_F, gallon, barrel, sec, ms, min, hr, deg, rad |

Table 2.1: **List of ALADDIN Keywords and Constants**

| Name/Argument | Purpose of Function |
|---|---|
| cos(x) | Compute cosine of quantity x. Default units are radians |
| sin(x) | Compute sin of quantity x. Default units are radians |
| tan(x) | Compute tangent of quantity x. Default units are radians |
| abs(x) | Return absolute value of quantity x. |
| exp(x) | Compute exponential of quantity x. |
| integer(x) | Return integer component of quantity x. |
| log(x) | Compute natural logarithm of quantity x. |
| log10(x) | Compute logarithm, base 10, of quantity x. |
| pow(x,n) | Compute quantity x raised to the power n; n is a number. |
| sqrt(x) | Compute square root of quantity x. |

Table 2.2: **List of Mathematical Functions**

## 2.4 Control of Program Flow

In ALADDIN, control of program flow is accomplished with logic operations on physical quantities, constructs for conditional branching, and looping constructs.

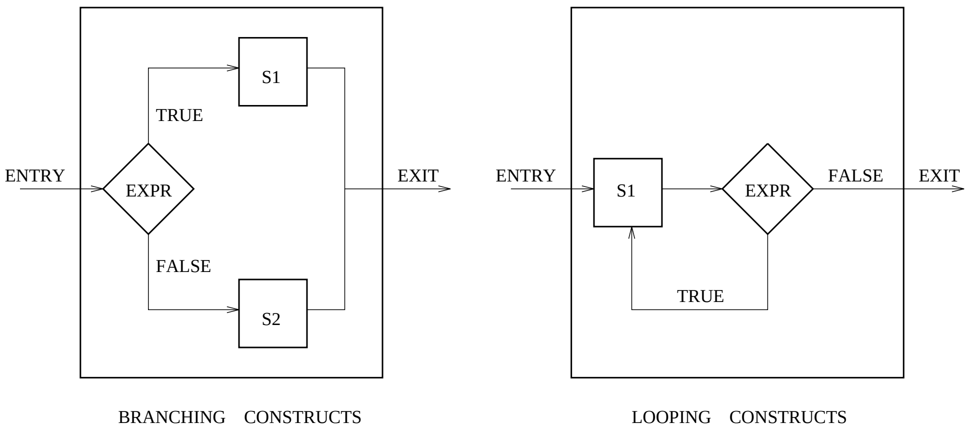


Figure 2.1: Branching and Looping Constructs in ALADDIN

Figure 2.1 is a schematic of branching and looping constructs. ALADDIN supports the `while` and `for` looping constructs, and `if` and `if-then-else` branching constructs.

### 2.4.1 Logical Operations

ALADDIN supports three logical operators on quantities – && means logical and; ! means logical not; and || means logical or. Operators for conditional branching include: `>` means greater than; `<` means less than; `>=` means greater than or equal to; `<=` means less thano or equal to; `==` means identically equal to. Examples of their use are dispersed throughout the following scripts of code.

INPUT | OUTPUT

```
X1 = 1  m; X2 = 10 m;

print "\n TEST RELATIONAL OPERATIONS \n";           TEST RELATIONAL OPERATIONS

Y1 = X1 > X2; Y2 = X1 < X2;

print" Y1 = ", Y1, " FALSE \n";                     Y1 =     0.0000e+00  FALSE
print" Y2 = ", Y2, " TRUE  \n";                     Y2 =     1.0000e+00  TRUE

Y1 = X1 >= X2; Y2 = X1 <= X2;

print" Y1 = ", Y1, " FALSE \n";                     Y1 =     0.0000e+00  FALSE
print" Y2 = ", Y2, " TRUE  \n";                     Y2 =     1.0000e+00  TRUE
```

```
Y1 = X1 == X2; Y2 = X1 != X2;

print" Y1 = ", Y1, " FALSE \n";                  Y1 =     0.0000e+00  FALSE
print" Y2 = ", Y2, " TRUE  \n";                  Y2 =     1.0000e+00  TRUE

print "\n TEST LOGICAL AND/OR \n";               TEST LOGICAL AND/OR

Y1 = (X1 == 1 m) && (X2 != 10 m);
Y2 = (X1 != 1 m) && (X2 == 10 m);
Y3 = (X1 == 1 m) && (X2 == 10 m);
Y4 = (X1 != 1 m) && (X2 != 10 m);

print" Y1 = ", Y1, " FALSE \n";                  Y1 =     0.0000e+00  FALSE
print" Y2 = ", Y2, " FALSE \n";                  Y2 =     0.0000e+00  FALSE
print" Y3 = ", Y3, " TRUE  \n";                  Y3 =     1.0000e+00  TRUE
print" Y4 = ", Y4, " FALSE \n";                  Y4 =     0.0000e+00  FALSE

Y1 = (X1 == 1 m) || (X2 != 10 m);
Y2 = (X1 != 1 m) || (X2 == 10 m);
Y3 = (X1 == 1 m) || (X2 == 10 m);
Y4 = (X1 != 1 m) || (X2 != 10 m);

print" Y1 = ", Y1, " TRUE  \n";                  Y1 =     1.0000e+00  TRUE
print" Y2 = ", Y2, " TRUE  \n";                  Y2 =     1.0000e+00  TRUE
print" Y3 = ", Y3, " TRUE  \n";                  Y3 =     1.0000e+00  TRUE
print" Y4 = ", Y4, " FALSE \n";                  Y4 =     0.0000e+00  FALSE
```

### 2.4.2 Conditional Branching

The `if` and `if-then-else` constructs allow for conditional branching of program flow. The command syntax is:

```
if(statement1) {        /* If the statements1 is true, then execute statements2 */
   statements2;
}                       /* Skip statements2 if statement1 is false             */
```

and

```
if(statement1) then {   /* If the statement1 is true, then execute statements2  */
   statements2;
} else {
   statements3;         /* If the statement1 is false, then execute statements3 */
}
```

with the braces "{" and "}" necessary even if `statements1` consists of a single statement. Here are two examples:

INPUT OUTPUT

```
print "\n -- if condition \n";                    -- if condition

x = 1 ksi;                                        x =       1 ksi
if ( x < 10 ksi ) {
     print " x = ", x ,"\n";
}

print "\n -- if-else condition \n";               -- if-else condition

x = 10 ksi; y = 1 MPa;
if ( x < 10 ksi ) then {
     print " x = ", x ,"\n";
} else {
     print " y = ", y ,"\n";                      y =       1 MPa
}
```

### 2.4.3 Looping and Stopping Commands

ALADDIN supports two looping constructs, the while-loop and the for-loop. The while-loop syntax is:

```
while (statement0) {
      statments;    /* If statement0 is true execute */
                    /* statements, else stop looping */
}
```

If `statement0` is true (i.e evaluates to a constant larger than zero). then the body of the while-loop (i.e `statements`) will be executed. Otherwise, the program will stop looping, and continue onto the next command. As we will see in the scripts of code below, `statement0` can be a single statement condition, or multiple conditions connected through logic operators.

The for-loop syntax is:

```
for ( initializer; condition; increment ){
      statements;
}
```

The `initializer`, `condition`, and `increment` statements can be either a series of quantity statements separated by comma (i.e ','), or, an empty statment. Zero or more statements may be located in the for-loop body.

The command for breaking one layer of loopings is `break` – and it must be used inside the looping body bounded by the symbol "{" and "}". The `quit` statement terminates program execution; it can be used anywhere outside the loops.

**While Loop with One Layer :** The examples of input commands for one-layer while-loop with different conditions are given below:

| INPUT | OUTPUT |
|---|---|

```
print "\n -- Single Condition \n";
X = 1 m;
while (X <= 5 m) {
   print " X = ", X, "\n";
   X = X + 1 m;
}
```

```
-- Single Condition
X =       1 m
X =       2 m
X =       3 m
X =       4 m
X =       5 m
```

```
print "\n -- Multiple Conditions \n";
Y = 1 in;
X = 1 m;
while (X <= 5 m && Y <= 0.5 ft) {
   print " (X, Y) = (",X,Y,")\n";
   X = X + 1 m;
   Y = Y + 1 in;
}
```

```
-- Multiple Conditions
(X, Y) = (       1 m       1 in)
(X, Y) = (       2 m       2 in)
(X, Y) = (       3 m       3 in)
(X, Y) = (       4 m       4 in)
(X, Y) = (       5 m       5 in)
```

```
print "\n -- Empty Condition \n";
X = 1 m;
while ( ) {
   if(X > 5 m) {
      break;
   }
   print " X = ", X, "\n";
   X = X + 1 m;
}
```

```
-- Empty Condition
X =       1 m
X =       2 m
X =       3 m
X =       4 m
X =       5 m
```

**While Loops with Multiple Layers :** The examples of input commands for multi-layers while-loop are given below:

| INPUT | OUTPUT |
|---|---|

```
print "\n -- Multiple Layers \n";
X = 2 m;
while (X <= 5 m) {
   Y = 1 in;
   print "\n";
   while(Y <= 1 ft) {
      print "(X, Y) = (",X,Y,")\n";
      Y = Y + 4 in;
   }
   X = X + 2 m ;
}
```

```
-- Multiple Layers
(X, Y) = (       2 m       1 in)
(X, Y) = (       2 m       5 in)
(X, Y) = (       2 m       9 in)
(X, Y) = (       4 m       1 in)
(X, Y) = (       4 m       5 in)
(X, Y) = (       4 m       9 in)
```

```
print "\n Break inside Multilayer While \n";

X = 2 m;
while (X <= 10 m) {
```

```
Break inside Multilayer While

(X, Y) = (       2 m       1 in)
(X, Y) = (       2 m       2 in)
```

```
    if(X < 4m) then {
    } else break;
    Y = 1 in;
    print "\n";
    while(Y <= 1 ft) {
        if(Y > 8 in)
           break;
        print "(X, Y) = (",X,Y,")\n";
        Y = Y + 1 in;
    }
    X = X + 2 m ;
}
```

```
(X, Y) = (       2 m       3 in)
(X, Y) = (       2 m       4 in)
(X, Y) = (       2 m       5 in)
(X, Y) = (       2 m       6 in)
(X, Y) = (       2 m       7 in)
```

---

**For Loops with One Layer :** The examples of input commands for one-layer for-loop with different conditions are given below:

INPUT

```
print "\n -- Empty Initializer\n";

x = 1m;
for(     ; x <= 5 m; x = x + 1 m) {
   print "x = ", x, "\n";
}

print "\n -- Empty Increment \n";

for(x = 1 m; x <= 5 m;       ) {
   print "x = ", x, "\n";
   x = x + 1 m;
}

print "\n -- Empty Condition\n";

for(x = 1 m;   ; x = x + 1 m ) {
   if(x > 5 m) {
      break;
   }
   print "x = ", x, "\n";
}

print "\n -- Empty Increment and Condition\n";

x = 1m;
for( ;  ;  ) {
    if(x > 5 m) {
      break;
    }
    print "x = ", x, "\n";
```

OUTPUT

```
-- Empty Initializer

x        = 1 m
x        = 2 m
x        = 3 m
x        = 4 m
x        = 5 m

-- Empty Increment

x        = 1 m
x        = 2 m
x        = 3 m
x        = 4 m
x        = 5 m

-- Empty Condition

x        = 1 m
x        = 2 m
x        = 3 m
x        = 4 m
x        = 5 m

-- Empty Increment and Condition

x        = 1 m
x        = 2 m
x        = 3 m
x        = 4 m
x        = 5 m
```

```
    x = x + 1m;
}

print "\n -- Single Condition \n";

for(x = 1 m; x <= 5 m; x = x + 1 m) {
   print "x = ", x, "\n";
}
```

```
-- Single Condition

x        = 1 m
x        = 2 m
x        = 3 m
x        = 4 m
x        = 5 m
```

```
print "\n -- Multiple Conditions \n";

x = 1 m;
y = 1 in;
z = 100 yard;

for(x = 1 m, y = 1 in, z = 100 yard; x <= 5 m
    && y < 1 ft || z < 0.5 mile; x = x + 1 m,
    y = y + 1 in, z = z + 400 yard) {
        print " (x,y,z) = (",x, y, z,")\n";
}
```

```
-- Multiple Conditions

(x,y,z) = (  1 m  1 in    100 yard)
(x,y,z) = (  2 m  2 in    500 yard)
(x,y,z) = (  3 m  3 in    900 yard)
(x,y,z) = (  4 m  4 in   1300 yard)
(x,y,z) = (  5 m  5 in   1700 yard)
```

The looping constructs `for` and `while` may be nested; examples are located in Chapter 4, and in the numerical and finite element algorithms developed for Part 2 of this report.

## 2.5 Definition and Printing of Matrices

The ALADDIN command language supports interactive definition of matrices, their allocation to variable names, and matrix operations. Matrix elements may be defined with physical units.

There are two basic mechanisms for defining a matrix; (a) build the matrix directly from an input command, and (b) make use of built-in matrix allocation functions,

### 2.5.1 Definition of Small Matrices

The following input commands demonstrate the interactive definition and printing of small matrices:

```
X = [1, 2, 3];
Y = [1; 2; 3];
Z = [1,
        3;
           2, 4];
```

```
PrintMatrix(X);
PrintMatrix(Y,Z);
```

X ,Y and Z are $(1 \times 3)$, $(3\times1)$ and $(2 \times 2)$ matrices, respectively. We use square brackets (i.e. `"["` and `"]"`) to indicate the beginning and end of a matrix definition. Individual elements of a matrix are separated by commas, and may be located on one or more lines of input. The elements of a matrix are defined row-wise, with a semi-colon positioned inside the brackets separating matrix rows. Each row of the matrix must have the same number of columns.

`PrintMatrix()` prints the contents of a matrix to standard output (i.e. the computer screen). With matrix X defined above, `PrintMatrix(X);` gives

```
MATRIX : "X"

row/col          1            2            3
      units
   1            1.00000e+00  2.00000e+00  3.00000e+00
```

Notice that a blank row and blank column are available for the printing of units – we will describe this feature in the next subsection. `PrintMatrix()` accepts from one to five matrix arguments. So, for example, the single command `PrintMatrix(Y, Z);` generates

```
MATRIX : "Y"
row/col          1
      units
   1            1.00000e+00
   2            2.00000e+00
   3            3.00000e+00

MATRIX : "Z"
row/col          1            2
      units
   1            1.00000e+00  3.00000e+00
   2            2.00000e+00  4.00000e+00
```

The same output would be generated by the command sequence:

```
PrintMatrix(Y); PrintMatrix(Z);
```

### 2.5.2 Built-in Functions for Allocation of Matrices

The interactive definition of matrices can be a error-prone process that becomes progressively tedious with increasing matrix size. In situations where numerical and finite element analysis problems generate matrices with hundreds – sometimes thousands – of rows and columns, the interactive definition of matrices is simply impractical. To

| Matrix | Purpose of Function |
|---|---|
| Matrix([s, t]) | Allocate s × t matrix |
| PrintMatrix(A, B,..) | Print matrices `A`, `B`, and so on. |
| ColumnUnits(A, [u]) | Assign column units `u` to matrix `A` – for complete details, see the text. |
| RowUnits(A, [u]) | Assign row units `u` to matrix `A` – for complete details, see the text. |
| Diag([s, n]) | Allocate s × s diagonal matrix with n along diagonal |
| One([s, t]) | Allocate s × t matrix full of ones |
| One([s]) | Allocate s × s matrix full of ones |
| Zero([s]) | Allocate s × s matrix full of zeros |
| Zero([s, t]) | Allocate s × t matrix full of zeros |

Table 2.3: **Functions for Definition and Printing of Matrices**

mitigate these limitations, ALADDIN provides a family of functions to generate matrices commonly used in numerical and finite element analyses.

Table 2.3 contains a summary of functions and their arguments for the definition of matrices (we will describe the finite element functions in Part 2 of this report). The following script of commented input code demonstrates and explains use of these functions.

START OF INPUT FILE

```
/* [a] : Allocate a 20 by 30 matrix              */

   W = Matrix([20, 30]);

/* [b] : Allocate a 1 by 30 matrix full of zeros */

   X = Zero([1, 30]);

/* [c] : Allocate a 30 by 30 matrix full of zeros */

   X = Zero([30, 30]);
   Y = Zero([30]);

/* [d] : Allocate a matrix full of zeros, the size is same of [W] */

   X = Zero(Dimension(W));

/* [e] : Allocate a 30 by 30 matrix full of ones */
```

```
   Y = One([30]);
   X = One([30, 30]);

/* [f] : Allocate a 30 by 30 diagonal matrix with 2 along diagonal     */
/*       and a 44 by 44 identity matrix                                 */

   X = Diag([30, 2]);
   Y = Diag([44, 1]);
```

---

### 2.5.3 Definition of Matrices with Units

Recall from Chapter 2 that matrix units are stored in row units and column units buffers. The definition of matrices with units falls into two classes, and we will demonstrate each by example.

**Row and Column Vectors :** Example commands for matrices that are either row or column vectors are

```
X = [1 kN, 2 Pa, 3 m];
Y = [1 kN; 2 Pa; 3 m];
Z = [lbf*ft; m^2; psi*in^2];
```

where the X, Y are 1x3, 3x1 matrices. The elements of X are `X[1][1] = 1.0 kN`, `X[1][2] = 2.0 Pa`, and `X[1][3] = 3 m`. The matrix Y is simply the transpose of matrix X. Matrix Z is a 3x1 matrix that consists of units alone. It's elements are `Z[1][1] = lbf.ft`, `Z[1][2] = m∧2` and `Z[1][3] = psi.in∧2 = lbf`.

**General Matrices :** For general matrices, units are written to units buffer with the built-in functions `ColumnUnits()` and `RowUnits()`. `ColumnUnits()` writes a column units buffer into a matrix. `RowUnits()` writes a row units buffer. `ColumnUnits()` and `RowUnits()` both accept either two or three arguments. The first argument is name of the matrix that units will be assigned to. The second argument is a $(1 \times n)$ units vector. The third argument is a $(1 \times 1)$ matrix containing the column number (or row number) units will be assigned to. In summary, the syntax is

```
W = ColumnUnits(M, [ units_vector ]);
W =    RowUnits(M, [ units_vector ]);

W = ColumnUnits(M, [ units_vector ], [ number ]);
W =    RowUnits(M, [ units_vector ], [ number ]);
```

**Example of ColumnUnits() :** Let `X` be a $(4 \times 4)$ non-dimensional matrix full of ones, possibly generated with the command `X = One([4,4]);`. The command

```
X = ColumnUnits(X, [kN]);
```

will assign the units kN to every element of the column units buffer in X. All of the elements of X will now have units kN. Two alternative ways of achieving the same result are:

```
X = ColumnUnits(X, [kN, kN, kN, kN]);
```

and

```
X = ColumnUnits(X, [kN], [1]);
X = ColumnUnits(X, [kN], [2]);
X = ColumnUnits(X, [kN], [3]);
X = ColumnUnits(X, [kN], [4]);
```

where the command `X = ColumnUnits(X, [kN], [1]);` assigns units kN to the first element of the column units buffer in X. It is important to note our careful choice of the words assign – when specific row or column numbers are omitted, units assignment takes place in all of those rows or columns that match the units exponents. Suppose, for example, that we generate X with the following sequence of statements:

```
X = One([6]);
X = ColumnUnits(X, [ton],  [1]);
X = ColumnUnits(X, [mile], [3]);
X = ColumnUnits(X, [klb],  [4]);
X = ColumnUnits(X, [ft],   [6]);

Y = ColumnUnits(X, [in, lb]);
```

The command `X = One([6])` generates a $(6 \times 6)$ matrix full of ones, and assigns the result to variable X. Repeated use of `ColumnUnits()` assigns to columns 1, 3, 4, 6 of X, units ton, mile, klb, and ft. Put another way, columns 1 and 4 have units of mass, and columns 3 and 6, units of length. Columns 2 and 5 are dimensionless. The details of X are:

```
MATRIX : "X"

row/col          1            2            3            4            5
      units         ton                       mile         klb
   1            1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00
   2            1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00
   3            1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00
   4            1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00
   5            1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00
   6            1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00


row/col          6
      units     ft
   1            1.00000e+00
   2            1.00000e+00
   3            1.00000e+00
   4            1.00000e+00
   5            1.00000e+00
   6            1.00000e+00
```

In the new matrix Y, the ton and klb are replaced with unit lb, so are the corresponding value of the elements, and miles and ft are replaced with in. The details of Y are:

```
MATRIX : "Y"

row/col          1            2            3            4            5
      units          lb                         in           lb
   1          2.00000e+06  1.00000e+00  6.33600e+04  1.00000e+03  1.00000e+00
   2          2.00000e+06  1.00000e+00  6.33600e+04  1.00000e+03  1.00000e+00
   3          2.00000e+06  1.00000e+00  6.33600e+04  1.00000e+03  1.00000e+00
   4          2.00000e+06  1.00000e+00  6.33600e+04  1.00000e+03  1.00000e+00
   5          2.00000e+06  1.00000e+00  6.33600e+04  1.00000e+03  1.00000e+00
   6          2.00000e+06  1.00000e+00  6.33600e+04  1.00000e+03  1.00000e+00

row/col          6
      units   in
   1          1.20000e+01
   2          1.20000e+01
   3          1.20000e+01
   4          1.20000e+01
   5          1.20000e+01
   6          1.20000e+01
```

### 2.5.4 Printing Matrices with Desired Units

The function PrintMatrixCast() prints a single matrix, or perhaps a family of matrices with desired units. The syntax for using PrintMatrixCast() is:

```
PrintMatrixCast( matrix1 , units_m );
PrintMatrixCast( matrix1 , matrix2, .... , units_m );
```

where matrix1, matrix2, and so on, are names of matrices to be printed, and units_m is a vector matrix containing the desired units. PrintMatrixCast() accepts from one to four arguments.

**Example :** Suppose that matrices X and Y are generated as follows

```
X = One([3, 4]);
Y = [2, 3; 4, 5];
X = ColumnUnits(X, [psi], [1]);
X = ColumnUnits(X, [kN],  [2]);
X = ColumnUnits(X, [km],  [4]);
X = RowUnits(X, [psi, N, mm]);

Y = ColumnUnits(Y, [psi]);
Y = RowUnits(Y,[in]);
```

The details of matrices X and Y are:

```
MATRIX : "X"

row/col          1            2            3            4
      units          psi          kN                        km
   1      psi   1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00
   2        N   1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00
   3       mm   1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00

MATRIX : "Y"

row/col          1            2
      units          psi          psi
   1       in   2.00000e+00  3.00000e+00
   2       in   4.00000e+00  5.00000e+00
```

Suppose that we now want to print `X` and `Y`, but with appropriate column units of pressure and length rescaled to `ksi` and `mm`. The command `PrintMatrixCast(X, Y, [ksi, mm]);` generates the output

```
MATRIX : "X"

row/col          1            2            3            4
      units          ksi          kN                        mm
   1      psi   1.00000e-03  1.00000e+00  1.00000e+00  1.00000e+06
   2        N   1.00000e-03  1.00000e+00  1.00000e+00  1.00000e+06
   3       mm   1.00000e-03  1.00000e+00  1.00000e+00  1.00000e+06

MATRIX : "Y"

row/col          1            2
      units          ksi          ksi
   1       in   2.00000e-03  3.00000e-03
   2       in   4.00000e-03  5.00000e-03
```

Row buffer units may also be re-scaled. For example, the command `PrintMatrixCast(X, Y, [ksi; mm]);` rescales the pressure and length units in appropriate rows to `ksi` and `mm` and generates the output:

```
MATRIX : "X"

row/col          1            2            3            4
      units          psi          kN                        km
   1      ksi   1.00000e-03  1.00000e-03  1.00000e-03  1.00000e-03
   2        N   1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00
   3       mm   1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00

MATRIX : "Y"

row/col          1            2
      units          psi          psi
   1       mm   5.08000e+01  7.62000e+01
   2       mm   1.01600e+02  1.27000e+02
```

## 2.6 Matrix-to-Quantity Conversion

Let `X` be a matrix, and `s` and `t` be positive integers. The command `Y = X[s][t]` extracts matrix element `X[s][t]` from `X`, and assigns the quantity to variable `Y`. An equivalent command for $(1 \times 1)$ matrices `X` is `Y = QuanCast(X)`. We demonstrate use of `QuanCast()` in the following script of code:

INPUT / OUTPUT

```
/* [a] : Define (1x1) matrix */

   X = [100 kPa];

/* [b] : Print element [1][1] of X */

   print "Y = ", QuanCast(X), "\n";             Y =      100 kPa
   print "Y = ", X[1][1]    , "\n";             Y =      100 kPa
```

## 2.7 Basic Matrix Operations

### 2.7.1 Retrieving the Dimensions of a Matrix

In the development of many numerical algorithms for engineering computations, there is often a need to extract the dimensions of a particular matrix. The number of rows and columns in a matrix may determine, for example, how many iterations of an algorithm will be computed. In ALADDIN, the function `Dimension( X )` extracts the number of rows and columns for matrix `X`, and returns the result in a $(1 \times 2)$ matrix. Elements `[1][1]` and `[1][2]` of the matrix returned by `Dimension()` contain the number of rows and columns in `X`, respectively.

INPUT / OUTPUT

```
/* [a] : Generate 13 by 20 matrix called Z */

   Z = One( [ 13, 20] );

/* [b] : Extract and print dimensions of Z */

   size = Dimension(Z);

   print "\n\n";
   print "Rows in [Z]    = ", size[1][1] ,"\n";  Rows in [Z]    =     1.3000e+01
   print "Columns in [Z] = ", size[1][2] ,"\n";  Columns in [Z] =     2.0000e+01
```

### 2.7.2 Matrix Copy and Matrix Transpose

The functions `Copy()` and `Transpose()` compute respectively, a matrix copy, and the matrix transpose. Both functions accept a single matrix argument. Examples of their use are located in the next subsection.

### 2.7.3 Matrix Addition, Subtraction, and Multiplication

Operations for basic matrix arithmetic include assignment, addition, subtraction, copying a matrix, and computing the matrix transpose and inverse. Here are some examples.

START OF INPUT FILE

```
/* [a] : define matrix A; use matrix copy and matrix transpose functions */

   A = [ 1.0 kN, 2.0 Pa, 3 cm ];
   B =  Copy(A);
   C = Trans(A);

/* [b] : define and print matrices X and Y, with units */

   X = Diag([4, 1]);
   Y = One([4]);
   X = ColumnUnits(X, [ksi, ft, N, m]);
   Y = ColumnUnits(Y, [psi, in, kN,km]);
   X = RowUnits(X, [psi, in, kN, mm]);
   Y = RowUnits(Y, [ksi, ft, N, mm]);

   PrintMatrix(X, Y);

/* [c] : compute matrix addition and subtraction, and print results */

   Z = X + Y; U = X - Y;

   PrintMatrix(Z, U);
```

Matrices `X` and `Y` are:

```
MATRIX : "X"

row/col          1            2            3            4
     units       ksi          ft           N            m
   1      psi   1.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00
   2       in   0.00000e+00  1.00000e+00  0.00000e+00  0.00000e+00
   3       kN   0.00000e+00  0.00000e+00  1.00000e+00  0.00000e+00
   4       mm   0.00000e+00  0.00000e+00  0.00000e+00  1.00000e+00

MATRIX : "Y"
```

| Name/Argument | Purpose of Function |
|---|---|
| Copy(A) | Make a copy of matrix `A`. |
| Dimension(A) | Retrieve the size, or dimensions, of matrix `A` – the result is a (1x2) matrix. |
| Extract(A,X,[s,t]) | Extract a submatrix of size `A` from matrix `X` starting at corner (s,t). |
| L2Norm(A) | Compute $L_2$ (Euclidean) norm of row or column vector `A`. |
| Put(X,A,[s,t]) | Put a p x q matrix A into matrix X, starting at corner (s,t). The contents of matrix X bounded by the row and column numbers (s,t) and (p+s-1,q+t-1) are replaced by the contents of A |
| QuanCast(A) | Cast a (1x1) matrix into a quantity. |
| Trans(A) | Compute transpose of matrix `A`. |

Table 2.4: **Functions for Basic Matrix Operations**

```
row/col          1            2            3            4
      units          psi          in           kN           km
   1      ksi   1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00
   2       ft   1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00
   3        N   1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00
   4       mm   1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00
```

Results of the matrix addition and subtraction are:

```
MATRIX : "Z"

row/col          1            2            3            4
      units          ksi          ft            N            m
   1      psi   2.00000e+00  8.33333e+01  1.00000e+06  1.00000e+06
   2       in   1.20000e-02  2.00000e+00  1.20000e+04  1.20000e+04
   3       kN   1.00000e-06  8.33333e-05  2.00000e+00  1.00000e+00
   4       mm   1.00000e-03  8.33333e-02  1.00000e+03  1.00100e+03

MATRIX : "U"

row/col          1            2            3            4
      units          ksi          ft            N            m
   1      psi   0.00000e+00 -8.33333e+01 -1.00000e+06 -1.00000e+06
   2       in  -1.20000e-02  0.00000e+00 -1.20000e+04 -1.20000e+04
   3       kN  -1.00000e-06 -8.33333e-05  0.00000e+00 -1.00000e+00
   4       mm  -1.00000e-03 -8.33333e-02 -1.00000e+03 -9.99000e+02
```

As pointed out in Chapter 2, consistent units are required for all matrix operations, including matrix addition and subtraction, and multiplication. Sometimes a power function can be used as a substitute to a series of matrix multiplications. Consider the following script of code to compute a square matrix `X` raised to the power of 3:

```
X = [3,  4, 5; 1, 6, 7; 81, 71,2];

Y = X*X*X;      /* triple product of matrices X */
Z = X^3;        /* X raised to the power of 3   */
```

The results are:

```
MATRIX : "Y"

row/col          1            2            3
      units
   1           5.93800e+03  7.78100e+03  4.93300e+03
   2           7.20600e+03  9.85700e+03  6.76100e+03
   3           7.57060e+04  7.15820e+04  1.04360e+04

MATRIX : "Z"

row/col          1            2            3
      units
   1           5.93800e+03  7.78100e+03  4.93300e+03
   2           7.20600e+03  9.85700e+03  6.76100e+03
   3           7.57060e+04  7.15820e+04  1.04360e+04
```

Notice that the syntax to compute a matrix raised to a power is the same as for a quantity raised to a power. Power operations may only be applied to certain classes of matrices:

[1] The matrix must be a square matrix – this requirement is needed because a matrix is multiplied from both left and right.

[2] The exponent is limited to integers larger than or equal to -1. Matrix raised to an exponent with fractional digits, such as $[X]^{0.3}$, or to a less than -1 power, such as $[X]^{-3}$ are lack of well defined mathematical meaning.

[3] The function is only useful for non-dimensional matrices. For dimensional matrices, two matrices can only multiply to each other when their units are consistent. And a matrix multiply to itself will result in unconsistent units.

A matrix raised to the power of zero is taken as a non-dimensional matrix full of ones.

### 2.7.4 Scaling a Matrix by a Quantity

Let `A` be a matrix and `q` a physical quantity. ALADDIN's grammar supports pre-multiplication of `A` by a quantity (i.e. `q.A`), post-multiplication of `A` by a quantity (i.e.

`A.q`), and division of `A` by a quantity (i.e. `A/q`). The following script of code demonstrates use of these commands. The script contains three parts – first, we generate a (4 × 4) matrix `X` with units. In Part 2, `X` is scaled by `2 in` via post-multiplication, and in Part 3, the elements of `X` are divided by `2 in`.

```
/* [a] : Define and print matrix [X] with units */

   X = One([4]);
   X = ColumnUnits(X, [ksi, lbf, ksi, ft]);
   X = RowUnits(X, [psi, in, kips, lb]);

   PrintMatrix(X);

/* [b] : Scale Matrix by constant factor of 2 in */

   print "\n Scale Matrix by constant factor of 2 in \n";

   Y = X*2 in;
   PrintMatrix(Y);

/* [c] : Reduce Matrix Content by a factor of 2 in */

   print "\n Reduce Matrix Content by a factor of 2 in \n";

   print "X [1][4] = ", X[1][4],"\n";
   Y = X/2 in;
   print "Y [1][4] = ", Y[1][4],"\n";
   PrintMatrix(Y);
```

The generated output is:

```
MATRIX : "X"

row/col          1            2            3            4
      units         ksi          lbf          ksi           ft
   1      psi   1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00
   2       in   1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00
   3     kips   1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00
   4       lb   1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00

Scale Matrix by constant factor of 2 in

MATRIX : "Y"

row/col          1            2            3            4
      units      lbf/in       lbf.in       lbf/in      in^ 2.0
   1      psi   2.00000e+03  2.00000e+00  2.00000e+03  2.40000e+01
   2       in   2.00000e+03  2.00000e+00  2.00000e+03  2.40000e+01
   3     kips   2.00000e+03  2.00000e+00  2.00000e+03  2.40000e+01
   4       lb   2.00000e+03  2.00000e+00  2.00000e+03  2.40000e+01

Reduce Matrix Content by a factor of 2 in
```

```
X [1][4] =        12 lbf/in
Y [1][4] =         6 psi

MATRIX : "Y"

row/col           1             2             3             4
      units   (ksi)/(in)       lbf/in   (ksi)/(in)
   1      psi   5.00000e-01  5.00000e-01  5.00000e-01  6.00000e+00
   2       in   5.00000e-01  5.00000e-01  5.00000e-01  6.00000e+00
   3     kips   5.00000e-01  5.00000e-01  5.00000e-01  6.00000e+00
   4       lb   5.00000e-01  5.00000e-01  5.00000e-01  6.00000e+00
```

Notice how the contents of `X[1][4]` and `Y[1][4]` have units that are the product of entries in the row and column units vectors.

### 2.7.5 Euclidean Norm of Row/Column Vectors

ALADDIN provides the function `L2Norm()` to compute the $L_2$ norm of either a $(1 \times n)$ row vector, or $(n \times 1)$ column vector.

$$\|x\|_2 = \sqrt{\left[\sum_{i=1}^{n} x_i^2\right]} \tag{2.1}$$

Consider the following example:

```
X = [1, 2, 3, 4, 5];

U = L2Norm(X);
PrintMatrix(X);

print "Euclidean Norm of [X] = ", U, "\n";
print "Euclidean Norm of [Y] = ", V, "\n";
```

The generated output is:

```
MATRIX : "X"

row/col           1             2             3             4             5
      units
   1            1.00000e+00  2.00000e+00  3.00000e+00  4.00000e+00  5.00000e+00

Euclidean Norm of [X] =     7.4162e+00
```

ALADDIN also allows $L_2$ norms to be computed for row and column vectors containing consistent sets of units. For example.

```
Y = [1 m; 2 m ; 3E-3 km; 4 m; 5 m];

V = L2Norm(Y);

PrintMatrix(Y);

print "Euclidean Norm of [Y] = ", V, "\n";
```

sets up a 1 by 5 row vector X, and a 5 by 1 column vector Y. Again, the generated output is:

```
MATRIX : "Y"

row/col          1
      units
   1        m   1.00000e+00
   2        m   2.00000e+00
   3       km   3.00000e-03
   4        m   4.00000e+00
   5        m   5.00000e+00

Euclidean Norm of [Y] =     7.4162e+00
```

Notice how the units have been stripped from the Euclidean norm.

### 2.7.6 Minimum and Maximum Matrix Elements

The matrix function `Min()` returns the minimum value of an element in a matrix, and the matrix function `Max()` returns the maximum value of an element in a matrix. Each function has one matrix argument, and returns a $(1 \times 1)$ matrix containing the min/max value. For example, the script of input

```
x = [ 3.78,  9.7, -4.7,  10.50  ;
      0.00, -5.8,  0.2,  -9.34] ;

MaxValue = Max(x);
MinValue = Min(x);

PrintMatrix(x);

print "Max(x) =", MaxValue ,"\n";
print "Min(x) =", MinValue ,"\n";
```

generates the output:

```
MATRIX : "x"

row/col                  1            2             3             4
        units
```

```
   1            3.78000e+00  9.70000e+00 -4.70000e+00  1.05000e+01
   2            0.00000e+00 -5.80000e+00  2.00000e-01 -9.34000e+00
Max(x) =      10.5
Min(x) =     -9.34
```

In this particular example the $(1 \times 1)$ matrices returned by `Max()` and `Min()` are converted to quantities when the assignments to `MaxValue` and `MinValue` are made. In engineering design applications, the `Min()` and `Max()` functions can simplify the writing of code where extreme values of system responses must be identified.

### 2.7.7 Substitution/Extraction of Submatrices

ALADDIN provides the function `Extract()` to extract a submatrix from a larger matrix, and the function `Put()` to replace a submatrix within a larger specified matrix. Both functions require three matrix arguments – for details, see Table 2.4. Suppose, for example, we generate matrices with the sequence of commands:

```
/* [a] : generate (3x4) matrix with units */

   X = One([3, 4]);
   X = ColumnUnits(X, [psi], [1]);
   X = ColumnUnits(X, [kN],  [2]);
   X = ColumnUnits(X, [km],  [4]);
   X = RowUnits(X, [psi, N, mm]);

/* [b] : Extract() a (2x2) matrix from a (3x4) matrix starting at corner (1,1) */

   A = Matrix([2,2]);
   A = Extract(A,X,[1,1]);

/* [c] : Extract() the 2nd column of matrix X  */

   B = Matrix([3,1]);
   B = Extract(B,X,[1,2]);

/* [d] : Extract() the 3rd row of matrix X     */

   C = Matrix([1,4]);
   C = Extract(C,X,[3,1]);

/* [e] : Put() a (2x2) submatrix D into X starting at corner (1,1) */

   D = [2 , 3 ; 6, 3];
   Y = Put(X, D, [1,1]);
```

The output from this script of code, including defined matrices X and D, and results of extracted matrices A,B,C and replaced matrix Y, is:

```
MATRIX : "X"

row/col          1            2            3            4
      units          psi           kN                      km
   1      psi   1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00
   2        N   1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00
   3       mm   1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00

MATRIX : "D"

row/col          1            2
      units
   1            2.00000e+00  3.00000e+00
   2            6.00000e+00  3.00000e+00

MATRIX : "A"

row/col          1            2
      units          psi            N
   1      psi   1.00000e+00  1.00000e+03
   2        N   1.00000e+00  1.00000e+03

MATRIX : "B"

row/col          1
      units
   1     psi.   1.00000e+00
   2   N^ 2.0   1.00000e+03
   3      N.m   1.00000e+00

MATRIX : "C"

row/col          1            2            3            4
      units       lbf/in          N.m           mm       m^ 2.0
   1            3.93701e-02  1.00000e+00  1.00000e+00  1.00000e+00

MATRIX : "Y"

row/col          1            2            3            4
      units
   1            2.00000e+00  3.00000e+00  1.00000e+00  1.00000e+00
   2            6.00000e+00  3.00000e+00  1.00000e+00  1.00000e+00
   3            1.00000e+00  1.00000e+00  1.00000e+00  1.00000e+00
```

## 2.8 Solution of Linear Matrix Equations

The purpose of this section is to describe the functions ALADDIN supports for the solution of linear equations

$$[A]\,\{x\} = \{b\} \tag{2.2}$$

We assume in equation (2.2) that $[A]$ is a $(n \times n)$ square matrix, and $\{x\}$ and $\{b\}$ are $(n \times 1)$ column vectors. Generally speaking, the computational work required to solve one or more families of linear equations is affected by: (a) the size and structure of matrix `A`, (b) the computational algorithm used to compute the numerical solution, and (c) the number of separate families of equations solutions are required.

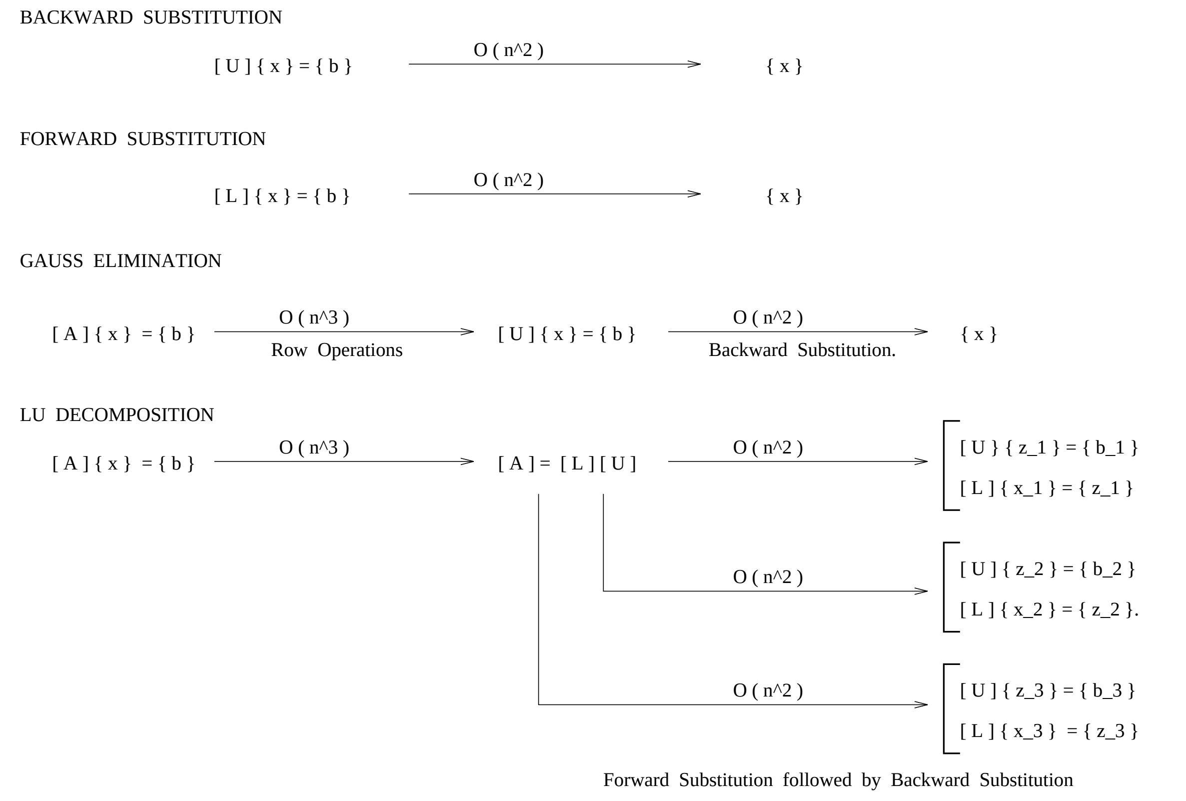


Figure 2.2: Strategies for Solving $[A]\,\{x\} = \{b\}$

Figure 2.2 summarizes four pathways of computation for the solution of linear equations $[A]\,\{x\} = \{b\}$. When matrix `A` is is either lower or upper triangular form, solutions to $[L]\,\{x\} = \{b\}$ can be computed with forward substitution, and solutions to $[U]\,\{x\} = \{b\}$ via backward substitution. Algorithms for forward/backward substitution require $O(n^2)$ computational work. The method of Guass Elimination is perhaps the most widely known method for solving systems of linear equations. In the first stage of Gauss Elimination,

| Name/Argument | Purpose of Function |
|---|---|
| Det(A) | Compute determinant of matrix `A` – the result is a quantity. |
| Decompose(A) | Compute LU decomposition for square matrix `A`. |
| Inverse(A) | Compute the inverse of matrix `A`. |
| Solve(A,b) | Compute solution to $[A]\{x\} = \{b\}$. |
| Substitution(LU,b) | Compute forward and backward substitution. LU is a matrix containing the LU decomposition of A. |

Table 2.5: **Functions for Solving Linear Equations**

a set of well defined row operations transforms $[A]\{x\} = \{b\}$ into $[U]\{x\} = \{b^*\}$. In the second stage of Gauss Elimination, the solution matrix `x` is computed via back substitution. The first and second stages of Gauss Elimination require $O(n^3)$ and $O(n^2)$ computational work, respectively.

In many numerical and finite element computations (e.g. structural dynamics with finite elements), families of equations [A]{x}= {b} are solved many times with different right-hand side vectors `b`. For example, in Figure 2.2, $\{b_1\}$, $\{b_2\}$, and $\{b_3\}$ represent three distinct right-hand sides to [A]{x}= {b}. The optimal solution procedure is to first decompose `A` into a product of lower and upper triangular matrices ($O(n^3)$ computational work), and then use forward substitution to solve $[L]\{z\} = \{b\}$, followed by backward substitution for $[U]\{x\} = \{z\}$ ($O(n^2)$ computational work). While solutions the first set of equations requires $O(n^3)$ computational work, solutions to all subsequent families of [A]{x} = {b's} requires only $O(n^2)$ computational work.

### 2.8.1 Solving $[A]\{x\} = \{b\}$

ALADDIN provides three functions, `Decompose()`, `Substitution()`, and `Solve()` for the solution of linear equations [A]{x}= {b}. A summary of ALADDIN's functions for solving linear equations is given in Table 2.5.

Function `Decompose()` computes the LU decomposition of matrix A. A typical example of its usage is:

```
LU = Decompose(A);
```

`Decompose()` will print an error message if matrix A is singular (i.e the determinant of A is zero, and a unique solution to the equations does not exist). With the LU decomposition of A complete, solutions to linear equations [L][U]{x}= {b} can be computed with forward and backward substitution. This two-step procedure is handled by a single call to `Substitution()`. A typical example of its usage is:

```
x = Substitution(LU, b).
```

As demonstrated in Chapter 1, solutions to linear equations may also be computed with the single command

```
x = Solve(A, b).
```

Function `Solve()` computes the solution to a single family of equations via the method of LU decomposition. The numerical procedure is identical to the two-command sequence

```
LU = Decompose(A);
x  = Substitution(LU, b).
```

LU decomposition can be used to solve a family of equations with different right-hand side vectors `b`. For example:

```
LU = Decompose(A);
x1  = Substitution(LU, b1).    /* Solve [A].x1 = b1 */
x2  = Substitution(LU, b2).    /* Solve [A].x2 = b2 */
x3  = Substitution(LU, b2).    /* Solve [A].x3 = b3 */
```

**Numerical Example :** Figure 2.3 shows a schematic and simplified model of an elastic cantilever beam. External loads include an axial force, a translational force (perpendicular to the axis of the beam), and a bending moment applied at the end point of the cantilever.

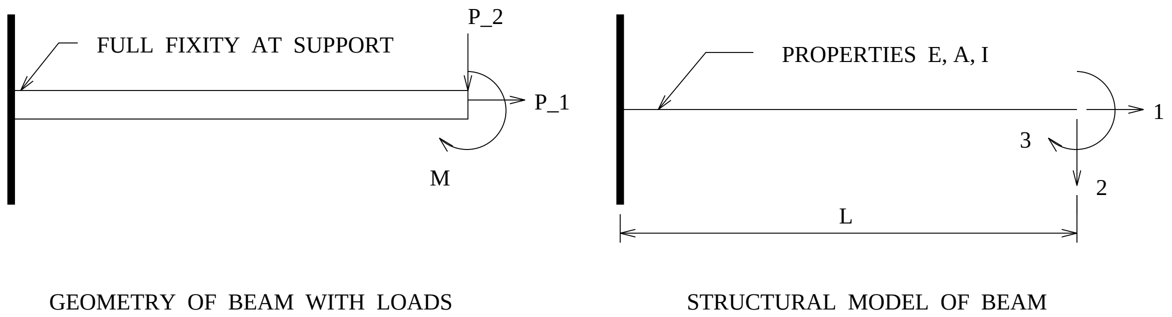


Figure 2.3: Schematic and Model of Prismatic Cantilever Beam

The cantilever is modeled with full fixity at its base. It has length $L = 5$ meters, cross section area $A = 0.10m^2$, moment of inertia $I = 0.10m^4$, and a Young's modulus of Elasticity $E = 200Pa$. The cross section shape and material properties are assumed to be constant along the length of the cantilever.

If deformations of the beam are small, and the material remains elastic, then the translational displacements and rotation of the cantilever end point are given by solutions to the equation

$$\begin{bmatrix} \frac{EA}{L} & 0 & 0 \\ 0 & \frac{12EI}{L^3} & \frac{-6EI}{L^2} \\ 0 & \frac{-6EI}{L^2} & \frac{4EI}{L} \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} = \begin{bmatrix} F_1 \\ F_2 \\ M \end{bmatrix} \tag{2.3}$$

In the first phase of our analysis, we define variables for properties of the cantilever beam, allocate memory for the stiffness matrix and two external load matrices, and initialize and print the contents of these matrices. These tasks are accomplished by the script of input file:

START OF INPUT FILE

```
/* [a] : Setup section/material properties for Cantilever Beam */

   E = 200   GPa;
   I = 0.10  m^4;
   A = 0.10  m^2;
   L = 5       m;

/* [b] : Define a (3x3) test matrix for Cantilever Beam */

   stiff = Matrix([3,3]);
   stiff = ColumnUnits( stiff, [N/m, N/m, N] );
   stiff =    RowUnits( stiff, [m], [3] );

   stiff[1][1] = E*A/L;
   stiff[2][2] = 12*E*I/(L^3);
   stiff[3][3] = 4*E*I/L;

   stiff[2][3] = -6*E*I/(L^2);
   stiff[3][2] = -6*E*I/(L^2);

   PrintMatrix(stiff);

/* [c] : Define and print two external load matrices */

   load1 = ColumnUnits( Matrix([3,1]), [N]);
   load1 = RowUnits( load1, [m], [3] );

   load1[1][1] =  10 kN;
   load1[2][1] =  10 kN;
   load1[3][1] =  00 kN*m;

   load2 = ColumnUnits( Matrix([3,1]), [N]);
   load2 = RowUnits( load2, [m], [3] );

   load2[1][1] =  10 kN;
   load2[2][1] = -10 kN;
   load2[3][1] =  00 kN*m;
```

```
   PrintMatrix( load1, load2 );
```

---

The contents of matrices `stiff`, `load1`, and `load2` are:

```
MATRIX : "stiff"

row/col          1            2            3
      units          N/m          N/m            N
   1          4.00000e+09  0.00000e+00  0.00000e+00
   2          0.00000e+00  1.92000e+09 -4.80000e+09
   3        m  0.00000e+00 -4.80000e+09  1.60000e+10

MATRIX : "load1"

row/col          1
      units
   1        N   1.00000e+04
   2        N   1.00000e+04
   3      N.m   0.00000e+00

MATRIX : "load2"

row/col          1
      units
   1        N   1.00000e+04
   2        N  -1.00000e+04
   3      N.m   0.00000e+00
```

Suppose that a unit displacement is applied to the $i^{th}$ degree of freedom in the structure, and displacements are zero at all other degrees of freedom. By definition, stiffness element $k_{ij}$ corresponds the force (or moment) required at degree of freedom `j` for equilibrium. In our cantilever structure, for example, the third row of the stiffness matrix corresponds to the forces needed produced a unit end rotation without also causing lateral or translational displacements. The required set of forces/moments are:

INPUT

```
   print "stiff[3][1] = ", stiff[3][1], "\n";
   print "stiff[3][2] = ", stiff[3][2], "\n";
   print "stiff[3][3] = ", stiff[3][3], "\n";
```

OUTPUT

```
stiff[3][1] =        0 N
stiff[3][2] = -4.8e+09 N
stiff[3][3] =  1.6e+10 N.m
```

---

Components `stiff[3][1]` to `stiff[3][3]` have units of force, force, and moment, respectively.

The linear equations are now solved in two ways. First, we use `Solve()` to compute displacements for load cases 1 and 2. Then we use `Decompose()` to compute the LU decomposition of `stiff`, followed by two applications of `Substitution()` for right-hand matrices `2*load1` and `2*load2`.

INPUT FILE CONTINUED

```
/* [d] : Use "Solve()" to compute solutions to sets of equations */

   displacements1 = Solve( stiff, load1 );
   PrintMatrix( displacements1 );

   displacements2 = Solve( stiff, load2 );
   PrintMatrix( displacements2 );

/* [e] : Use "Decompose()" followed by multiple calls to "Substitution()" */

   lu = Decompose( stiff );

   displacements1 = Substitution( lu, 2*load1 );
   displacements2 = Substitution( lu, 2*load2 );

   PrintMatrix( displacements1 );
   PrintMatrix( displacements2 );
```

For ease reading, displacements `1` and `2`, and `3` and `4` are juxtaposed.

```
MATRIX : "displacements1"                MATRIX : "displacements2"

row/col           1                      row/col           1
      units                                    units
   1        m   2.50000e-06              1        m   2.50000e-06
   2        m   2.08333e-05              2        m  -2.08333e-05
   3            6.25000e-06              3           -6.25000e-06

MATRIX : "displacements3"                MATRIX : "displacements4"

row/col           1                      row/col           1
      units                                    units
   1        m   5.00000e-06              1        m   5.00000e-06
   2        m   4.16667e-05              2        m  -4.16667e-05
   3            1.25000e-05              3           -1.25000e-05
```

Before the forward and backward substitution proceeds, ALADDIN checks that the units of `LU` and `b` are compatible, and automatically assigns units to the solution vector. For our cantilever example, degrees of freedom one and two correspond to axial and translational displacements of the cantilever end point. They have units of length. Degree of freedom three is rotation of the cantilever end point – it is printed as a dimensionless quantity.

Finally, we check the accuracy of our numerical solution by computing the residual of solutions:

INPUT FILE CONTINUED

```
/* [e] : Compute and printk residuals of solution matrices */

   R1 = stiff*displacements1 -   load1;
   R2 = stiff*displacements2 -   load2;
   R3 = stiff*displacements3 - 2*load1;
   R4 = stiff*displacements4 - 2*load2;

   PrintMatrix( R1, R2, R3, R4 );
   quit;
```

generates the output

```
MATRIX : "R1"                                   MATRIX : "R2"

row/col          1                              row/col          1
      units                                           units
   1        N    0.00000e+00                    1        N    0.00000e+00
   2        N    0.00000e+00                    2        N    0.00000e+00
   3      N.m    0.00000e+00                    3      N.m    0.00000e+00

MATRIX : "R3"                                   MATRIX : "R4"

row/col          1                              row/col          1
      units                                           units
   1        N    0.00000e+00                    1        N    0.00000e+00
   2        N    0.00000e+00                    2        N    0.00000e+00
   3      N.m    0.00000e+00                    3      N.m    0.00000e+00
```

### 2.8.2 Matrix Inverse

The inverse of matrix A is denoted $A^{-1}$, and for non-dimensional matrices, is given by solutions to the matrix equations:

$$A \cdot A^{-1} = A^{-1} \cdot A = I. \tag{2.4}$$

For matrices containing units, the left- and right-inverses are given by

$$A \cdot A^{-1} = \left(A^{-1} \cdot A\right)^T = IL, \tag{2.5}$$

and

$$A^{-1} \cdot A = \left(A \cdot A^{-1}\right)^T = IR. \tag{2.6}$$

In Chapter 7 we will see that numerical values of matrix elements of IR and IL are symmetric, but the left- and right- identity matrices have units that are transposed with respect to one another. Also, when the units are removed from IR and IL, IR = IL = I.

An inverse to matrix A will exist if and only if A is nonsingular (i.e. the determinant of A is nonzero). ALADDIN employs a two step method for computing the inverse of a matrix:

[1] Compute the LU decomposition of A. If the product of diagonal terms in either `L` or `U` equals zero, then `A` is singular, and an inverse does not exist.

[2] Let $b_i$ correspond to the $i^{th}$ column of `I`. For each $b_i$, i = 1 to $n$, compute forward and backward substitution for $[L][U]\{x_i\} = \{b_i\}$. Solution $\{x_i\}$ corresponds to the $i^{th}$ column of $A^{-1}$.

The following script of code demonstrates two ways of computing the inverse of a matrix `X`.

START OF INPUT FILE

```
/* [a] Define (4x4) matrix X */

   X = [ 3,  4,  5, 7;
         1,  6,  7, 9;
        11, 26, 47, 9;
        81, 71,  2, 2 ];

/* [b] Compute inverse of X in two ways */

   Y = Inverse(X);
   Z = X^-1;

   PrintMatrix(X, Y, Z);

/* [c] Compute and print left and right identity matrices */

   IL = X*Y;    /* Left  identity matrix */
   IR = Y*X;    /* Right identity matrix */

   PrintMatrix(IL, IR);
```

Notice how the product of matrices `X` and `Y` is computed by simply typing `X*Y`. The matrices X, Y, IR, and IL are

```
MATRIX : "X"

row/col          1            2            3            4
      units
   1         3.00000e+00  4.00000e+00  5.00000e+00  7.00000e+00
   2         1.00000e+00  6.00000e+00  7.00000e+00  9.00000e+00
   3         1.10000e+01  2.60000e+01  4.70000e+01  9.00000e+00
   4         8.10000e+01  7.10000e+01  2.00000e+00  2.00000e+00

MATRIX : "Y"
```

```
row/col          1            2            3            4
      units
   1            3.50926e-01 -2.77316e-01  3.87423e-03  2.24593e-03
   2           -4.07636e-01  3.19090e-01 -4.66030e-03  1.17911e-02
   3            1.16086e-01 -1.15216e-01  2.63616e-02 -6.45705e-03
   4            1.42476e-01  1.88097e-02 -1.78271e-02 -3.08815e-03

MATRIX : "Z"

row/col          1            2            3            4
      units
   1            3.50926e-01 -2.77316e-01  3.87423e-03  2.24593e-03
   2           -4.07636e-01  3.19090e-01 -4.66030e-03  1.17911e-02
   3            1.16086e-01 -1.15216e-01  2.63616e-02 -6.45705e-03
   4            1.42476e-01  1.88097e-02 -1.78271e-02 -3.08815e-03

MATRIX : "IL"

row/col          1            2            3            4
      units
   1            1.00000e+00 -1.94289e-16  0.00000e+00 -3.46945e-18
   2           -2.22045e-16  1.00000e+00  2.77556e-17  3.46945e-18
   3            6.66134e-16 -5.27356e-16  1.00000e+00 -2.42861e-17
   4            3.05311e-15 -7.45931e-15  2.77556e-17  1.00000e+00

MATRIX : "IR"

row/col          1            2            3            4
      units
   1            1.00000e+00 -5.82867e-16 -3.22659e-16 -6.83481e-16
   2            3.33067e-16  1.00000e+00  3.29597e-16  1.07553e-16
   3            1.11022e-16 -5.55112e-17  1.00000e+00 -1.04083e-16
   4           -5.55112e-17 -2.77556e-17 -6.67869e-17  1.00000e+00
```

For non-dimensional matrices, the left and right identity products $X \cdot X^{-1}$ and $X^{-1} \cdot X$ are identical (except for roundoff errors). Now let's look at the inverse of square matrices having units – here is one example.

START OF INPUT FILE

```
/* [a] Define (4x4) matrix X */

   X = [ 3,  4,  5;
         1,  6,  7;
        81, 71,  2 ];

   X = ColumnUnits(X, [N/m, N/m, N]);
   X = RowUnits(X, [m], [3]);

/* [b] Compute inverse of X in two ways */

   Y  = X^-1;
```

```
   PrintMatrix(X, Y);

/* [c] Compute and print left and right identity matrices */

   IL = X*Y;    /* Left  identity matrix */
   IR = Y*X;    /* Right identity matrix */

   PrintMatrix(IL, IR);
```

The generated output is

```
MATRIX : "X"

row/col          1            2            3
      units          N/m          N/m            N
   1            3.00000e+00  4.00000e+00  5.00000e+00
   2            1.00000e+00  6.00000e+00  7.00000e+00
   3        m   8.10000e+01  7.10000e+01  2.00000e+00

MATRIX : "Y"

row/col          1            2            3
      units                            m^-1.0
   1      m/N   3.81890e-01 -2.73228e-01  1.57480e-03
   2      m/N  -4.44882e-01  3.14173e-01  1.25984e-02
   3   N^-1.0   3.26772e-01 -8.74016e-02 -1.10236e-02

MATRIX : "IL"

row/col          1            2            3
      units          N/m          N/m           Pa
   1      m/N   1.00000e+00 -5.55112e-17 -6.93889e-18
   2      m/N   4.44089e-16  1.00000e+00  0.00000e+00
   3  Pa^-1.0  -6.66134e-16 -1.36002e-15  1.00000e+00

MATRIX : "IR"

row/col          1            2            3
      units          N/m          N/m            N
   1      m/N   1.00000e+00 -1.94289e-16 -8.67362e-17
   2      m/N   0.00000e+00  1.00000e+00 -2.22045e-16
   3   N^-1.0   0.00000e+00 -1.11022e-16  1.00000e+00
```

## 2.9 Matrix Eigenvalues and Eigenvectors

Let $\mathbf{K}$ be a $(n \times n)$ matrix. The eigenvector of $\mathbf{K}$ is defined as a vector, which when premultiplied by $\mathbf{K}$ generates a vector proportional to the original vector. The constant of proportionality is called the eigenvalue. In mathematical notation we write

$$\mathbf{K}\phi = \lambda\phi, \tag{2.7}$$

where $\phi$ is the $(n \times 1)$ eigenvector, and $\lambda$ the eigenvalue. We say that equation (2.7) is an eigenvalue problem when it is solved for the $\lambda's$. And it is an eigenvector problem when (2.7) is solved for the $\phi's$. Equation (2.7) is an eigenproblem when it is solved for both the $\lambda's$ and $\phi's$.

In this section we describe the two types of eigenproblems ALADDIN can solve. The first, and simpler case, is the standard eigenproblem

$$\mathbf{K}\Phi = \Phi\Lambda. \tag{2.8}$$

In equation (2.8), $\mathbf{K}$ is a $(n \times n)$ positive definite symmetric stiffness matrix, and $\Lambda$ and $\Phi$ are $(n \times n)$ matrices of eigenvalues and eigenvectors, respectively (i.e. $\Phi = [\phi_1, \phi_2, \cdots, \phi_n]$), and $\Lambda = diag(\lambda_1, \lambda_2, \cdots, \lambda_n)$. ALADDIN also supports solutions to the generalized eigenproblem

$$\mathbf{K}\Phi = \mathbf{M}\Phi\Lambda, \tag{2.9}$$

where $\mathbf{M}$ is a $(n \times n)$ positive definite matrix, and the remaining symbols take their aforementioned definition. Generalized eigenproblems play a fundamental role in the free-vibration analysis of structural and mechanical systems, studies of instability in engineering systems, and in the identification of principal moments of inertia in two- and three-dimensional solids.

In the analysis of many engineering systems, only the first $p$ $(p < n)$ eigenpairs are needed for analysis. The solution for $p$ eigenpairs in the standard eigenproblem corresponds to

$$\mathbf{K}\Phi = \Phi\Lambda. \tag{2.10}$$

The definition of $\Phi$ is modified to a $(n \times p)$ matrix containing the first $p$ eigenvectors (i.e. $\Phi = [\phi_1, \phi_2, \cdots, \phi_p]$), and $\Lambda = diag(\lambda_1, \lambda_2, \cdots, \lambda_p)$ is a $(p \times p)$ diagonal matrix containing the first $p$ eigenvalues. The counterpart to equation (2.9) is

$$\mathbf{K}\Phi = \mathbf{M}\Phi\Lambda. \tag{2.11}$$

| Name/Argument | Purpose of Function |
|---|---|
| Eigen(K,M,[s]) | Compute `s` eigenvalues and eigenvectors. Result of the function call is assigned to `eigen`. |
| Eigenvalue(eigen) | Extract eigenvalues from array `eigen`. |
| Eigenvector(eigen) | Extract eigenvectors from array `eigen`. |

Table 2.6: **Eigenvalue and Eigenvector Functions**

In equations (2.10) and (2.11), it is important to notice how the eigenvalues and eigenvectors have been carefully positioned so that both the left- and right-hand sides evaluate to a $(n \times p)$ matrix.

ALADDIN employs a subspace iteration algorithm to solve equations (2.8) and (2.11). A theoretical discussion of the algorithm may be found in Bathe [7]. Details of the C code may be found in Austin and Mazzoni [5].

### 2.9.1 Solving $\mathbf{K\Phi = M\Phi\Lambda}$

ALADDIN provides three functions, `Eigen()`, `Eigenvalue()`, and `Eigenvector()`, for the solution of the matrix eigenvalue problems – a summary of these functions and their arguments is given in Table 2.6. `Eigen()` requires three matrix arguements as input. The first and second arguments are the stiffness and mass matrices, and the third argument, a $(1 \times 1)$ matrix containing the number of eigenvalues and eigenvectors the subspace algorithm should compute. Here is skeleton script of code for the solution of an eigenproblem

```
no_eigen = 2;
eigen = Eigen( K, M, [ no_eigen ]);
eigenvalue  = Eigenvalue( eigen );
eigenvector = Eigenvector( eigen );
```

If $\mathbf{K}$ and $\mathbf{M}$ are $(n \times n)$ matrices, then `eigen` is a $((n+1) \times n)$ matrix containing a composite of eigenvalues and eigenvectors. Functions `Eigenvalue()` and `Eigenvector()` simply extract the appropriate rows and columns from `eigen` – the same result could be achieved with `Extract()`.

We now apply these functions to two applications, elastic buckling of a pin-ended column, and vibration analysis of a supported cantilever beam.

**Numerical Example 1 :** Figure 2.4 shows a pin-ended column acted on by a central axial compressive force P. The column has length $L = 5m$, modulus of elasticity $E = 200GPa$,

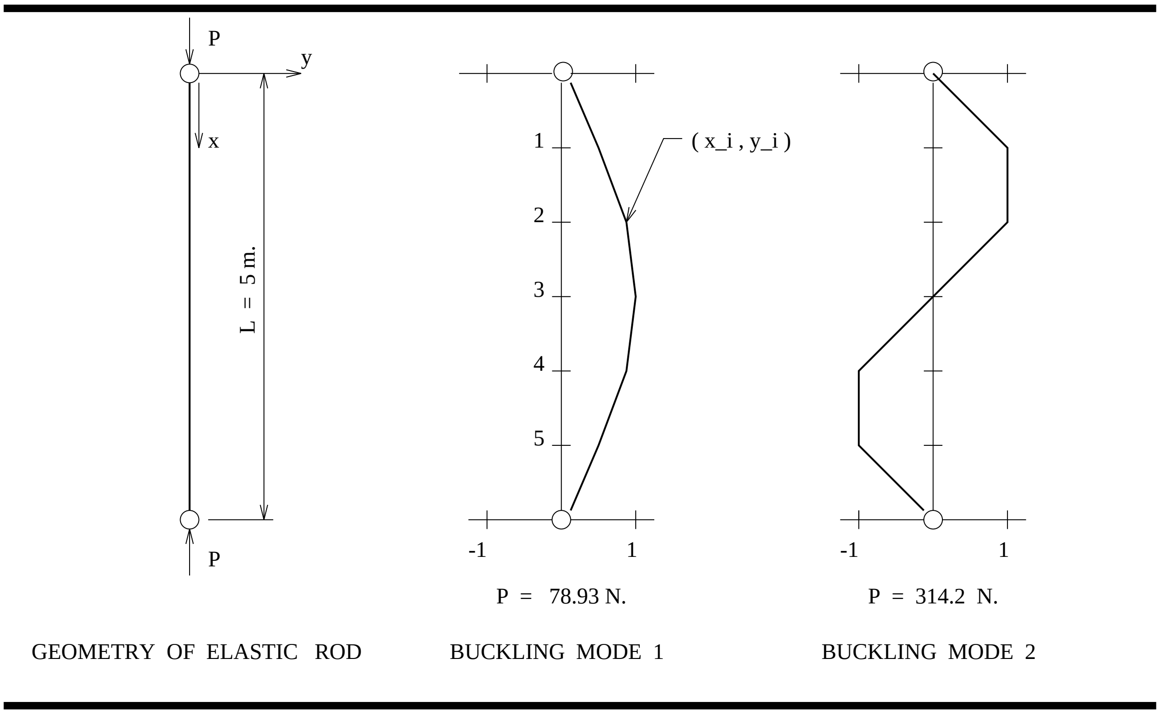


Figure 2.4: Elastic Buckling of Pin-Ended Column

and constant moment of inertia $I = 1000mm^4$ along its length. The theoretical buckling load is given by solutions to

$$EI \cdot \frac{d^2x}{d^2y} + P.y = 0 \tag{2.12}$$

with boundary conditions $y(0) = y(L) = 0$. Solutions to equation (2.12) are well known, and may be found in most undergraduate texts on structural mechanics – hence, we will simply state the results here. The deformed column will be in static equilibrium, and satisfy the boundary conditions, when

$$P_n = \frac{n^2\pi^2 EI}{L^2} \quad \text{for } n = 1, 2, 3, \cdots. \tag{2.13}$$

The mode shape $Y_n(x)$ corresponding to critical load $P_n$ is

$$Y_n(x) = A.\sin\left(\frac{n\pi x}{L}\right). \tag{2.14}$$

For our example column, first and second mode buckling loads and modal shapes are:

$$P_1 = \frac{\pi^2 EI}{L^2} = 78.957N \qquad Y_1(x) = \sin\left(\frac{\pi x}{5}\right), \tag{2.15}$$

and

$$P_2 = \frac{4\pi^2 EI}{L^2} = 315.82N \qquad Y_2(x) = \sin\left(\frac{2\pi x}{5}\right). \tag{2.16}$$

Numerical estimates of the column buckling loads and shapes of deformation may be computed via a finite difference approximation to equation (2.12), followed by solution of the ensuing eigenproblem. We will illustrate the numerical method by computing the column's lateral displacements at five equally spaced nodal points $(x_1, y_1)$, $(x_2, y_2)$, $(x_3, y_3)$, $(x_4, y_4)$ and $(x_5, y_5)$, as shown in the center diagram of Figure 2.4.

For step one, Ghali and Neville [14] have shown that a suitable finite difference approximation to equation (2.12) is

$$\begin{bmatrix} -1 & 2 & -1 \end{bmatrix} \cdot \begin{bmatrix} y_{(i-1)} \\ y_i \\ y_{(i+1)} \end{bmatrix} = \frac{P_{cr}\xi^2}{12EI} \begin{bmatrix} 1 & 10 & 1 \end{bmatrix} \cdot \begin{bmatrix} y_{(i-1)} \\ y_i \\ y_{(i+1)} \end{bmatrix} \tag{2.17}$$

where coordinates $y_{(i-1)}$, $y_{(i)}$, and $y_{(i+1)}$ are lateral displacements of the column, and $\xi$ is the x coordinate spacing between nodes (i.e. $\xi = x_{(i+1)} - x_i = L/6$). Global behavior of the column is given by the sequential application of equation (2.17) to each of the five internal nodes. The eigenproblem that results is

$$[A]\{y\} = \lambda [B]\{y\}, \quad \text{where} \quad \lambda = \frac{P_{cr}\xi^2}{12EI}, \tag{2.18}$$

and `A` and `B` are $(5 \times 5)$ matrices, and $\{y\}$ is a $(5 \times 1)$ matrix of lateral displacements along the column. The details of `A` and `B` are as follows

$$A = \begin{bmatrix} 2 & -1 & 0 & 0 & 0 \\ -1 & 2 & -1 & 0 & 0 \\ 0 & -1 & 2 & -1 & 0 \\ 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & -1 & 2 \end{bmatrix} \quad B = \begin{bmatrix} 10 & 1 & 0 & 0 & 0 \\ 1 & 10 & 1 & 0 & 0 \\ 0 & 1 & 10 & 1 & 0 \\ 0 & 0 & 1 & 10 & 1 \\ 0 & 0 & 0 & 1 & 10 \end{bmatrix}. \tag{2.19}$$

.

**Details of Input File :** A four-part input file is needed for the definition of column geometry, material and section properties, assembly of the finite difference matrices, solution of the eigenvalue problem, and printing of the buckling loads and modal shapes.

START OF INPUT FILE

```
/* [a] : Define section/material properties Buckling Problem */

   E = 200   GPa;
   I = 1000 mm^4;
   L = 5       m;

/* [b] : Define a (5x5) matrices for finite difference approximation */
```

```
   A = [  2, -1,  0,  0,  0;
         -1,  2, -1,  0,  0;
          0, -1,  2, -1,  0;
          0,  0, -1,  2, -1;
          0,  0,  0, -1,  2 ];

   B = [ 10,  1,  0,  0,  0;
          1, 10,  1,  0,  0;
          0,  1, 10,  1,  0;
          0,  0,  1, 10,  1;
          0,  0,  0,  1, 10 ];

   PrintMatrix(A,B);

/* [c] : Compute Eigenvalues and Eigenvectors */

   no_eigen    = 2;
   eigen       = Eigen(A, B, [ no_eigen ]);
   eigenvalue  = Eigenvalue(eigen);
   eigenvector = Eigenvector(eigen);

/* [d] : Print Eigenvalues, Eigenvectors, and Buckling Loads */

   size = Dimension(A);
   for(i = 1; i <= no_eigen; i = i + 1) {
       print "\n";
       print "Mode", i ,"\n";
       print "Eigenvalue      = ", eigenvalue[i][1],                 "\n";
       print "Buckling Load P = ", 12*E*I*eigenvalue[i][1]/(L/6)^2 ,"\n";

       for(j = 1; j <= size[1][1]; j = j + 1) {
           if(j == 1) then {
             print "\n";
             print "Mode Shape :", eigenvector[j][i], "\n";
           } else {
             print "            ", eigenvector[j][i], "\n";
           }
       }
   }
   print "\n";
```

**Abbreviated Output File :** We have used the parameter `no_eigen` to control the number of eigenvalues/vectors that are computed by the subspace iteration, and printed.

START OF ABBREVIATED OUTPUT FILE

```
MATRIX : "A"

row/col                 1            2            3            4            5
        units
   1          2.00000e+00 -1.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00
   2         -1.00000e+00  2.00000e+00 -1.00000e+00  0.00000e+00  0.00000e+00
```

```
   3            0.00000e+00 -1.00000e+00  2.00000e+00 -1.00000e+00  0.00000e+00
   4            0.00000e+00  0.00000e+00 -1.00000e+00  2.00000e+00 -1.00000e+00
   5            0.00000e+00  0.00000e+00  0.00000e+00 -1.00000e+00  2.00000e+00

MATRIX : "B"

row/col                  1            2            3            4            5
        units
   1            1.00000e+01  1.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00
   2            1.00000e+00  1.00000e+01  1.00000e+00  0.00000e+00  0.00000e+00
   3            0.00000e+00  1.00000e+00  1.00000e+01  1.00000e+00  0.00000e+00
   4            0.00000e+00  0.00000e+00  1.00000e+00  1.00000e+01  1.00000e+00
   5            0.00000e+00  0.00000e+00  0.00000e+00  1.00000e+00  1.00000e+01

*** SUBSPACE ITERATION CONVERGED IN 10 ITERATIONS

Mode         1
Eigenvalue      =    0.02284
Buckling Load P =      78.93 N

Mode Shape :       0.5
                 0.866
                     1
                 0.866
                   0.5

Mode         2
Eigenvalue      =    0.09091
Buckling Load P =      314.2 N

Mode Shape :         1
                0.9988
             -0.001186
               -0.9988
               -0.9976
```

A summary of numerical results is shown on the right-hand side of Figure 2.4. The numerical results for both the buckling loads and modal shapes are in very good agreement with equations (2.15) and (2.16).

**Numerical Example 2 :** The purposes of this numerical example are to demonstrate use of:

[1] Element level stiffness and mass matrices, as would be found in finite element analysis.

[2] Destination arrays as a means of mapping degrees of freedom in the mass and stiffness finite element matrices onto the global stiffness matrix (and mass matrix) degrees of freedom.

[3] The eigenvalue functions to compute the natural periods, and modal shapes of vibration.

We have deliberately selected a problem with a small number of unknowns so that step-by-step details of items [1] and [2] may be written down, and solved using functions from the matrix library alone.

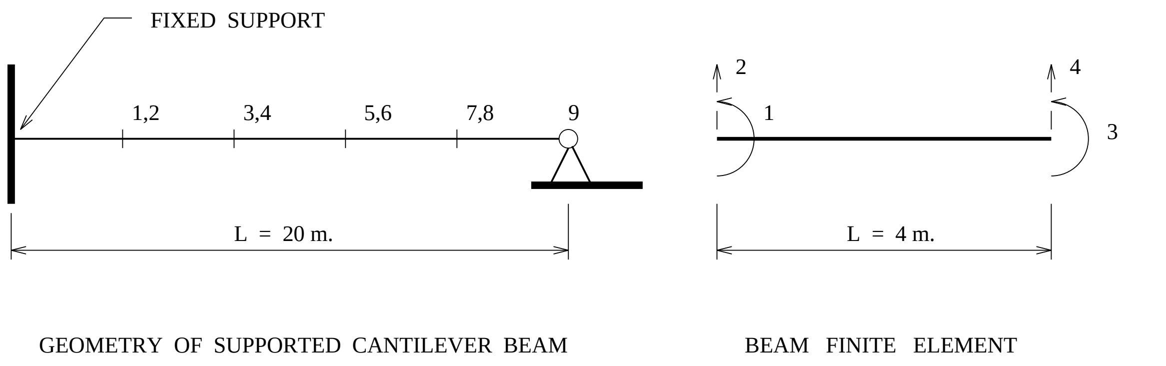


Figure 2.5: Single Span Cantilever Beam with Supported End Point

Figure 2.5 shows a uniform cantilever beam with a supported end point. The cantilever has length 20 m, modulus of elasticity $E = 200\ GPa$, and constant moment of inertia $I = 15.5 \times 10^6\ mm^4$ along its length. We will assume that the cantilever is rigid in its axial direction (i.e. beam cross section area $\approx \infty\ m^2$), and that all other deformations are small. The boundary conditions are full-fixity at the base of the cantilever, and zero translational displacement at the cantilever end.

Overall behavior of the cantilever will be modeled with five beam finite elements, and nine global degrees of freedom. They are a translation and rotation at each of the internal nodes, and a single rotational degree of freedom at the hinged cantilever support. Numbering for the global degrees of freedom is shown on the left-hand side of Figure 2.5. Lateral displacements along each beam element will be expressed as the superposition of four cubic interpolation functions, each weighted by a nodal displacement factor. The four nodal displacements are two beam-end lateral displacements, and two beam-end rotations, as shown on the right-hand side of Figure 2.5.

**Element Stiffness and Mass Matrices :** It is well known that if cubic interpolation functions are used to describe displacements along the beam element, then the beam element stiffness matrix will be

$$\mathbf{K} = \left[\frac{2EI}{L}\right] \cdot \begin{bmatrix} 2 & 3/L & 1 & -3/L \\ 3/L & 6/L^2 & 3/L & -6/L^2 \\ 1 & 3/L & 2 & -3/L \\ -3/L & -6/L^2 & -3/L & 6/L^2 \end{bmatrix}, \tag{2.20}$$

where EI is the flexural rigidity of the beam element, and L its length. We say that a mass matrix is "consistent" when the shape functions used to evaluate the mass matrix are the same as those for the stiffness matrix. The consistent mass matrix is

$$\mathbf{M} = \left[\frac{\bar{m}L}{420}\right] \cdot \begin{bmatrix} 4L^2 & 22L & -3L^2 & 13L \\ 22L & 156 & -13L & 54 \\ -3L^2 & -13L & 4L^2 & -22L \\ 13L & 54 & -22L & 156 \end{bmatrix}, \tag{2.21}$$

where $\bar{m}$ is average mass per unit length. Notice that in the translational degrees of freedom (i.e. columns 2 and 4 and rows 2 and 4 of **M**), the coefficients 156, 54, 156, and 54 sum to 420, thereby giving a total beam mass $\bar{m}L$.

**Assembly of Global Stiffness and Mass Matrices :** The assembly process for the global stiffness and global mass matrices may be written

$$\mathbf{K} = \sum_{i=1}^{N} K_i \quad \text{and} \quad \mathbf{M} = \sum_{i=1}^{N} M_i \tag{2.22}$$

where $K_i$ and $M_i$ represent the stiffness and mass matrices for beam element `i` mapped onto the global degrees of freedom, and `N` is the total number of elements in the model. A straight forward way of assembling these matrices is with the use of destination arrays. The destination array for our cantilever problem is

$$\mathbf{LD} = \begin{bmatrix} 0 & 0 & 1 & 2 \\ 1 & 2 & 3 & 4 \\ 3 & 4 & 5 & 6 \\ 5 & 6 & 7 & 8 \\ 7 & 8 & 9 & 0 \end{bmatrix}. \tag{2.23}$$

Element `(i,j)` of **LD** contains the global degree of freedom for degree of freedom `j` of beam finite element `i`. A zero entry indicates that the beam element degree of freedom will not be mapped to the global stiffness matrix. So, for example, degrees of freedom 1 and 2 of beam element 1 are not mapped to the global stiffness matrix – this is because the left-most section of the cantilever is firmly attached to the wall. Degrees of freedom 3 and 4 of beam element 1 are mapped to degrees of freedom 1 and 2 in the global stiffness and mass matrices. Similarly, degrees of freedom 1, 2, and 3 of beam element 5 will be mapped to global degrees of freedom 7, 8, and 9. Degree of freedom 4 of beam element 5 is not mapped to the global stiffness matrix because lateral displacements of the cantilever end are prevented by the hinged support.

**Details of Input File :** A finite element model for the supported cantilever is defined in parts [a] to [f] of the following input file. The eigenvalue problem is solved in part [g].

ABBREVIATED INPUT FILE

```
/* [a] : Define section/material properties */

   E    = 200000   MPa;
   I    = 15.5E+6 mm^4;
   L    = 4       m;
   mbar = 31.6 kg/m;

/* [b] : Define (4x4) stiffness matrix for beam element */

   stiff = Matrix([4,4]);
   stiff = ColumnUnits( stiff, [N,  N/m,  N,  N/m] );
   stiff =    RowUnits( stiff, [m], [1] );
   stiff =    RowUnits( stiff, [m], [3] );

   stiff[1][1] =  4*E*I/L;
   stiff[1][2] =  6*E*I/(L^2);
   stiff[1][3] =  2*E*I/L;
   stiff[1][4] = -6*E*I/(L^2);

   stiff[2][1] =   6*E*I/(L^2);
   stiff[2][2] =  12*E*I/(L^3);
   stiff[2][3] =   6*E*I/(L^2);
   stiff[2][4] = -12*E*I/(L^3);

   stiff[3][1] =  2*E*I/L;
   stiff[3][2] =  6*E*I/(L^2);
   stiff[3][3] =  4*E*I/L;
   stiff[3][4] = -6*E*I/(L^2);

   stiff[4][1] =  -6*E*I/(L^2);
   stiff[4][2] = -12*E*I/(L^3);
   stiff[4][3] =  -6*E*I/(L^2);
   stiff[4][4] =  12*E*I/(L^3);

   PrintMatrix(stiff);

/* [c] : Define (4x4) consistent mass matrix for beam element */

   mass = Matrix([4,4]);
   mass = ColumnUnits( mass, [ kg*m, kg, kg*m, kg] );
   mass =    RowUnits( mass, [m], [1] );
   mass =    RowUnits( mass, [m], [3] );
```

```
   mass[1][1] = (mbar*L/420)*4*L*L;
   mass[2][1] = (mbar*L/420)*22*L;
   mass[3][1] = (mbar*L/420)*-3*L^2;
   mass[4][1] = (mbar*L/420)*13*L;

   mass[1][2] = (mbar*L/420)*22*L;
   mass[2][2] = (mbar*L/420)*156;
   mass[3][2] = (mbar*L/420)*-13*L;
   mass[4][2] = (mbar*L/420)*54;

   mass[1][3] = (mbar*L/420)*-3*L^2;
   mass[2][3] = (mbar*L/420)*-13*L;
   mass[3][3] = (mbar*L/420)*4*L^2;
   mass[4][3] = (mbar*L/420)*-22*L  ;

   mass[1][4] = (mbar*L/420)*13*L;
   mass[2][4] = (mbar*L/420)*54;
   mass[3][4] = (mbar*L/420)*-22*L;
   mass[4][4] = (mbar*L/420)*156;

   PrintMatrix(mass);

/* [d] : Destination Array beam element connectivity */

   LD = [ 0, 0, 1, 2  ;
          1, 2, 3, 4  ;
          3, 4, 5, 6  ;
          5, 6, 7, 8  ;
          7, 8, 9, 0 ];

/* [e] : Allocate memory for global mass and stiffness matrices */

   GMASS = Matrix([9,9]);
   GMASS = ColumnUnits( GMASS, [ kg*m, kg, kg*m, kg, kg*m, kg, kg*m, kg, kg*m ] );
   GMASS =    RowUnits( GMASS, [m], [1] );
   GMASS =    RowUnits( GMASS, [m], [3] );
   GMASS =    RowUnits( GMASS, [m], [5] );
   GMASS =    RowUnits( GMASS, [m], [7] );
   GMASS =    RowUnits( GMASS, [m], [9] );

   GSTIFF = Matrix([9,9]);
   GSTIFF = ColumnUnits( GSTIFF, [ N, N/m,  N,  N/m, N, N/m, N, N/m, N ] );
   GSTIFF =    RowUnits( GSTIFF, [m], [1] );
   GSTIFF =    RowUnits( GSTIFF, [m], [3] );
   GSTIFF =    RowUnits( GSTIFF, [m], [5] );
   GSTIFF =    RowUnits( GSTIFF, [m], [7] );
   GSTIFF =    RowUnits( GSTIFF, [m], [9] );

/* [f] : Assemble Global Stiffness/Mass Matrices for Two Element Cantilever */

   no_elements = 5;
   for( i = 1; i <= no_elements; i = i + 1) {
   for( j = 1; j <= 4; j = j + 1) {
```

```
   row = LD [i][j];
   if( row > 0) {
       for( k = 1; k <= 4; k = k + 1) {
          col = LD [i][k];
          if( col > 0) {
              GMASS [ row ][ col ] = GMASS [ row ][ col ] + mass[j][k];
              GSTIFF[ row ][ col ] = GSTIFF[ row ][ col ] + stiff[j][k];
          }
       }
   }
   }
   }

/* [g] : Compute and Print Eigenvalues and Eigenvectors */

   no_eigen    = 2;
   eigen       = Eigen( GSTIFF, GMASS, [ no_eigen ]);
   eigenvalue  = Eigenvalue(eigen);
   eigenvector = Eigenvector(eigen);

   for(i = 1; i <= no_eigen; i = i + 1) {
       print "Mode", i ," : w^2 = ", eigenvalue[i][1];
       print " : T = ", 2*PI/sqrt(eigenvalue[i][1]) ,"\n";
   }

   PrintMatrix(eigenvector);
```

Points to note are:

[1] Element level stiffness matrices are stored in `stiff`. Rows 1 and 3 of `stiff` represent equations of equilibrium when a moment is applied at the beam end to cause a unit rotation, and rows 2 and 4, equations of equilibrium for unit lateral displacements of the beam element. We distinguish these cases by applying units of `m` to rows 1 and 3.

[2] Element level mass matrices are stored in `mass`. By definition, terms in the mass matrix correspond to forces, or torques, required to cause unit translational or rotational acceleration. A unit acceleration in a translational degree of freedom will require a force of the type *force* = *mass* × *acc'n*, with mass taking kilogram (kg) units. A unit rotational acceleration will have the form *torque* = *inertia* × *angular acc'n*. The units for *inertia* are `kg` $m^2$.

[3] The $(9 \times 9)$ global mass and global stiffness matrices are stored in `GMASS` and `GSTIFF`, respectively. The statements `row = LD[i][j]` and `col = LD[i][j]` contain the row-column mapping for the beam element degrees of freedom onto the global degrees of freedom.

**Abbreviated Output File :** Details of the beam element stiffness and mass matrices, and ensuing eigenvalues and eigenvectors are as follows:

START OF ABBREVIATED OUTPUT FILE

```
MATRIX : "stiff"

row/col          1            2            3            4
      units            N          N/m            N          N/m
   1        m   3.10000e+06  1.16250e+06  1.55000e+06 -1.16250e+06
   2            1.16250e+06  5.81250e+05  1.16250e+06 -5.81250e+05
   3        m   1.55000e+06  1.16250e+06  3.10000e+06 -1.16250e+06
   4           -1.16250e+06 -5.81250e+05 -1.16250e+06  5.81250e+05

MATRIX : "mass"

row/col          1            2            3            4
      units         kg.m           kg         kg.m           kg
   1        m   1.92610e+01  2.64838e+01 -1.44457e+01  1.56495e+01
   2            2.64838e+01  4.69486e+01 -1.56495e+01  1.62514e+01
   3        m  -1.44457e+01 -1.56495e+01  1.92610e+01 -2.64838e+01
   4            1.56495e+01  1.62514e+01 -2.64838e+01  4.69486e+01

*** SUBSPACE ITERATION CONVERGED IN  7 ITERATIONS

Mode    1.0000e+00  : w^2 =    145.8 rad.sec^-2.0 : T =   0.5203 sec
Mode    2.0000e+00  : w^2 =     1539 rad.sec^-2.0 : T =   0.1602 sec

MATRIX : "eigenvector"

row/col          1            2
      units
   1            1.24900e-01 -2.28500e-01
   2        m   3.02715e-01 -7.71256e-01
   3            1.03061e-01  1.77895e-01
   4        m   8.01556e-01 -9.46803e-01
   5           -1.29310e-02  3.44764e-01
   6        m   1.00000e+00  3.03399e-01
   7           -1.37281e-01 -5.51890e-02
   8        m   6.87189e-01  1.00000e+00
   9           -1.89640e-01 -3.58198e-01
```

Points to note are:

[1] The eigenvalues of this problem correspond to the circular natural frequency squared of the individual modes, and they take the units `rad/sec/sec`. The natural period of vibration for the first and second modes is `0.5203` seconds, and `0.1602` seconds, respectively.

[2] The first two eigenvectors are summarized in the script of abbreviated output. Perhaps the most surprising result of this analysis is the unit on the eigenvectors – for

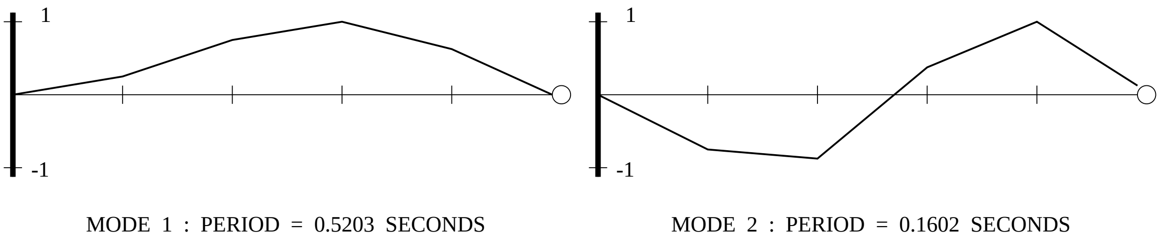


Figure 2.6: Mode Shapes and Periods of Vibration for Single Span Steel Beam

those degrees of freedom corresponding to translational displacements, the eigenvector is printed with units of length. The remaining degrees of freedom represent rotations, and take the (non-dimensional) units of radians.

[3] Figure 2.6 shows the mode shapes and natural periods of vibration for the supported cantilever beam. We have drawn the mode shapes as connected straight lines – actually, the mode shapes will be cubic splines, as defined by the beam element shape functions.

# Chapter 3

# Construction of Numerical Algorithms

## 3.1 Introduction

The purpose of this chapter is to test the flexibility of ALADDIN's command language by constructing basic numerical algorithms for computing the roots of nonlinear equations, and the minima of multi-dimensional functions subject to linear and nonlinear constraints. We have selected this problem domain because the numerical algorithms may be incorporated into finite element analyses of linear and nonlinear structures. For details, see Chen and Austin [10].

## 3.2 Roots of Nonlinear Equations

Let $x = (x_1, x_2, \cdots, x_n)^T$ by a coordinate point in n-dimensional space, and $f = (f_1, f_2, \cdots, f_n)^T$ be a n-dimensional (nonlinear) vector. The roots of nonlinear equations $f$ are given by solutions to

$$f(x) = 0. \tag{3.1}$$

### 3.2.1 Newton-Raphson and Secant Algorithms

The methods of `Newton-Raphson` and `Secant Approximation` are among the most popular for computing the roots of nonlinear equations. Let vector $x_o$ be an initial guess at the solution to equation (3.1). The method of Newton-Raphson corresponds to a sequence of first order Taylor series expansion about $x_{(k)}$, namely

$$f(x_{(k+1)}) = f(x_{(k)}) + \nabla f(x_{(k)}) \cdot (x_{(k+1)} - x_{(k)}) \approx 0$$

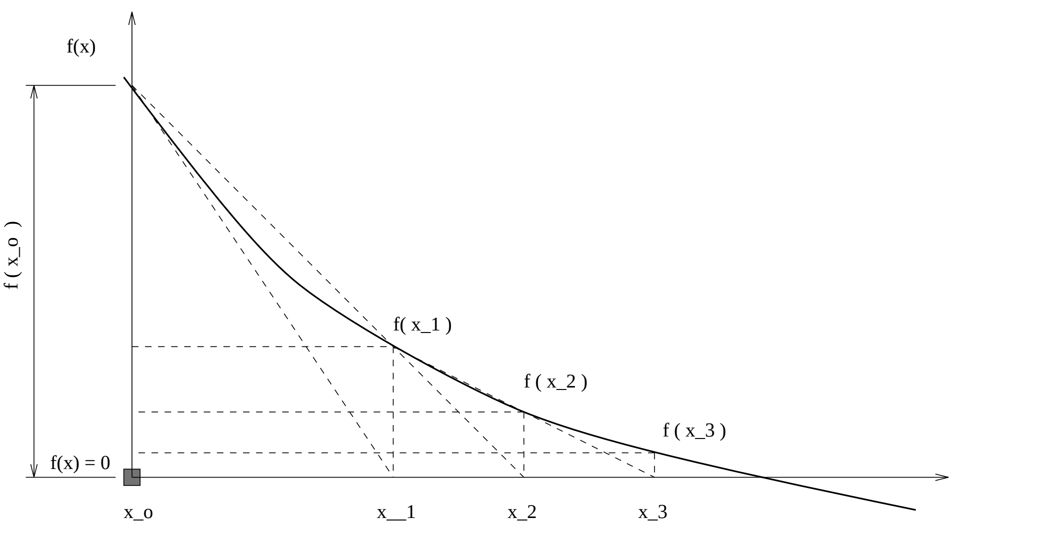


Figure 3.1: Secant Approximation of Quasi-Newton Method

where k = 0, 1, 2, $\cdots$, n, and $(\nabla f(x))_{ij} = (\partial f_i)/(\partial x_j)$. In matrix form, the gradient approximation is

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f_1(x)}{\partial x_1} & \frac{\partial f_1(x)}{\partial x_2} & \cdots & \frac{\partial f_1(x)}{\partial x_n} \\ \frac{\partial f_2(x)}{\partial x_1} & \frac{\partial f_2(x)}{\partial x_2} & \cdots & \frac{\partial f_2(x)}{\partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial f_n(x)}{\partial x_1} & \frac{\partial f_n(x)}{\partial x_2} & \cdots & \frac{\partial f_n(x)}{\partial x_n} \end{bmatrix}. \tag{3.2}$$

If $\nabla f(x_{(k)})$ has an inverse, then the (full) Newton-Raphson update is:

$$x_{(k+1)} = x_{(k)} - \Big[f(x_{(k)})\Big]^{-1} \cdot \nabla f(x_{(k)}). \tag{3.3}$$

The procedure is repeated until convergence criteria are satisfied. While the method of full Newton-Raphson can be efficient in some specific nonlinear analyses, Bathe [7, 9] reports that in general, full Newton-Raphson is not a competitive computational method for computing the roots of a wide-range of nonlinear equations. A major limitation of full Newton-Raphson is the need for updating and factorizing the coefficient matrix -$\nabla f(x_{(k)})$ at each iteration. One strategy for avoiding these computationally expensive steps is to replace -$\nabla f(x_{(k)})$ in equation (3.3) with -$\nabla f(x_{(0)})$, thereby eliminating the need to recalculate and factorize -$\nabla f(x_{(k)})$ at each iteration. From a mathematical point of view, this simplification corresponds to a linearization of the gradient $f(x_o)$. For problems with significant nonlinearities – in particular when the system stiffens during the response – this linearization can lead to a very slow convergence in the iteration. Even worse, the iteration may diverge.

The class of quasi-Newton methods are a second alternative to full Newton Raphson. Quasi-Newton methods update the coefficient matrix, or it's inverse, to provide a secant approximation to the matrix from iteration `(k)` to `(k+1)`. Figure 3.1 shows, for example, a sequence of secant approximations one might apply during the computation of roots in a one-dimension nonlinear equation. If a displacement increment is defined

$$\delta_{(k+1)} = x_{(k+1)} - x_{(k)} \tag{3.4}$$

an increment in the residuals defined as

$$\gamma_{(k+1)} = f(x_{(k)}) - f(x_{(k+1)}), \tag{3.5}$$

then the updated secant stiffness matrix, $K_{(k+1)}$, will satisfy the quasi-Newton equation

$$K_{(k+1)}\delta_{(k+1)} = \gamma_{(k+1)}$$

.

### 3.2.2 Broyden-Fletcher-Goldfarb-Shanno (BFGS) Algorithm

Among Quasi-Newton methods, the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method appears to be the most effective. Within iteration `(k+1)`, the BFGS method employs the following procedure to evaluate $x_{(k+1)}$ and $K_{(k+1)}$ [9]:

**Step 1 :** Initialization. Provide an initial value of vector $x = x_0$, and calculate the corresponding $K_o$ matrix at $x = x_o$.

**Step 2 :** Evaluated the x vector increment

$$\Delta x = -\Big[f(x_{(k)})\Big]^{-1} \cdot \nabla f(x_{(k)}). \tag{3.6}$$

This vector increment defines a *direction* for the actual increment.

**Step 3 :** Perform a line-search along direction $\Delta x$ to satisfy "equilibrium." Details of the line-search are as follows. First, set $\beta = 1.0$ and let the x vector

$$x_{(k+1)} = x_{(k)} + \beta \Delta x \tag{3.7}$$

where $\beta$ is a scalar multipler. The product $\beta \mid \Delta x \mid$ represents the distance between points $x_{(k)}$ and $x_{(k+1)}$. The value of $\beta$ is changed until the component of the residual in direction $\Delta x$, as defined by the inner product of $\Delta x^T f(x_{(k+1)})$, is approximately zero. The latter condition is satisfied when

$$\Delta x^T \cdot f(x_{(k+1)}) \leq \; STOL \cdot \Delta x^T \cdot f(x_{(k)}). \tag{3.8}$$

STOL is a convergence tolerance. The distance of the line search is automatically halved after each failure of equation (3.8) by halving $\beta$; $x_{(i)}$ is then recalculated with the new

$\beta$. If after five or ten line search attempts, equation (3.8) still isn't satisfied, then the direction given by the $\Delta x$ is probablely wrong. A new direction calculation is needed. This procedure is summarized in four-substeps:

**Step 3.1 :** Use equation (3.6) to update `x`.

**Step 3.2 : `Begin while loop`** : While equation (3.8) is not satisfied, work through Substeps 3.3 to 3.5.

**Step 3.3 :** If the line search number is less than or equal to `MaxLineSearchCount`, then set $\beta = \beta/2.0$, and use equation (3.6) to update `x`.

**Step 3.4 :** If the line search number is greater than `MaxLineSearchCount`, then break the while loop. Recalculate matrix `K` at $x = x_{(k+1)}$. Go to Step 2.

**Step 3.5 : `End while loop`** : Go to Step 4 for BFGS update.

**Step 4 :** Use equations (3.4) and (3.5) to calculate $\delta_{(k+1)}$ and $\gamma_{(k+1)}$.

**Step 5 :** Use the BFGS update [9] to revise the inverse of coefficient matrix `K`. The update of `K` is given by

$$K^{-1}{}_{(k+1)} = A_{(k+1)}{}^{T} K^{-1}{}_{(k)} A_{(k+1)} \tag{3.9}$$

where the matrix $A_{(k+1)}$ is a $(n \times n)$ matrix

$$A_{(k+1)} = I + v^{(k+1)} \cdot w^{(k+1)^T}. \tag{3.10}$$

Vectors $v^{(k+1)}$ and $w^{(k+1)}$ are given by

$$v^{(k+1)} = -\left[ \frac{\delta_{(k+1)}{}^{T} \gamma_{(k+1)}}{\delta_{(k+1)}{}^{T} \beta f(x_{(k)})} \right] \cdot \beta \cdot f(x_{(k)}) - \gamma_{(k+1)} \tag{3.11}$$

and

$$w^{(k+1)} = \left[ \frac{\delta_{(k+1)}}{\delta_{(k+1)}{}^{T} \gamma_{(k+1)}} \right]. \tag{3.12}$$

**Step 6 :** To avoid numerically dangerous updates, the conditional number

$$c_{(k+1)} = \left[ \frac{\delta_{(k+1)}{}^{T} \gamma_{(k+1)}}{\delta_{(k+1)}{}^{T} \beta f(x_{(k)})} \right]^{1/2} \tag{3.13}$$

of the updating matrix $A_{(k+1)}$ must be compared to some predetermined tolerance. A large condition number implies that the updated inverse matrix will be nearly singular. Numerical updates are not performed if the condition number exceeds this tolerance – in this project, we follow the recommendation of Bathe [9], and set the tolerance at $10^5$.

**Step 7 :** Check convergence of *force* and *energy* equilibriums. The force convergence criterion requires that the norm of the out-of-balance residual or *force* to be within a pre-set tolerance $\epsilon_F$ of the first residual.

$$\|f(x_{(k+1)})\|_2 \leq \epsilon_F \cdot \|f(x_{(o)})\| \tag{3.14}$$

A force criterion is not sufficient to ensure convergence. Consider the case of function f(x) with small or closing to zero gradients, the out-of-balance residual or *force* may be very small while the variable x or displacement may be grossly in error. Therefore, the "energy equilibrium" condition is necessary to provide the indication that both $\delta x^{(k+1)}$, and residual are approaching zeros. It requires computation of the work done by the force residual moving through displacement increment $\Delta x$.

$$\Delta x_{(k+1)} \cdot f(x_{(k)}) \leq \epsilon_E \cdot \Delta x_{(o)} \cdot f(x_{(0)}) \tag{3.15}$$

where the $\epsilon_E$ is a preset energy tolerance. In the following examples, we will use $\epsilon_F = 10^{-4}$ and $\epsilon_E = 10^{-5}$.

**Numerical Example :** We demonstrate use of the BFGS algorithm by solving a two-variable family of nonlinear equations:

$$\begin{aligned} 2{x_1}^2 + x_2 &= 4 \\ x_1 + 2{x_2}^2 &= 5 \end{aligned} \tag{3.16}$$

These equations may also be written $\{F\} - \{R\} = 0$, where

$$\{F\} = \left\{ \begin{array}{c} 2{x_1}^2 + x_2 \\ x_1 + 2{x_2}^2 \end{array} \right\} \quad \{R\} = \left\{ \begin{array}{c} 4 \\ 5 \end{array} \right\} \tag{3.17}$$

We use the parameter `MaxLineSearchCount = 5` for the maximum number for line search trials, and select an initial trial vector $\{x\} = (7, 10)^T$.

START OF INPUT FILE

```
/*
 *  ===================================================
 *  Use BFGS Algorithm to Solve Nonlinear Equations
 *
 *  Function :
 *              2x[1]^2 + x[2] = 4
 *              x[1] + 2x[2]^2 = 5
 *
 *          R^t     = (4, 5)
 *          F[1]    = 2x[1]^2 + x[2]
 *          F[2]    = x[1] + 2x[2]^2
 *          K[i][j] = dF[i]/dx[j]          Jacobain matrix
 *
 *  Written By : X.G. Chen                        June 1994
 *  ===================================================
```

```
 */

dimen = 2;
IMAX  = 20;     /* Maximum number of iteration */
Ef    = 1E-4;
Ee    = 1.E-5;
beta  = 1.0;
STOL  = 0.5;
EPS   = 1.E-5;
ii    = 1;
MaxLineSearchCount = 5;

/*  [a] : Initialize and print problem parameters and initial solution */

    x = [7; 10];
    R = [4; 5];

    print "------initial Guess -----\n";
    PrintMatrix(x);
    print "\n------Given R value -----\n";
    PrintMatrix(R);

    Y       = [2*x[1][1], 1; 1, 2*x[2][1]];
    K       = [4*x[1][1], 1; 1, 4*x[2][1]];
    F       = Y*x;
    Residu0 = R-F;
    Residu  = R-F;
    K_inver = Inverse(K);

/*  [b] : Allocate working matrices */

    Residu_old   = Residu;
    I            = Diag([dimen, 1]);
    delta_old    = Zero([dimen, 1]);

/*  [c] : Main Loop for Iteration */

    Iter_no = 0;
    for (ii = 1; ii <= IMAX; ii = ii + 1) {

        Iter_no = Iter_no + 1;
        beta    = 1.0;
        delta   = K_inver*Residu;

     /* [c.1] : Armijo Line Search */

        x_temp  = x + beta*delta;
        temp1   = QuanCast(Trans(delta)*Residu);
        Y       = [2*x_temp[1][1], 1; 1, 2*x_temp[2][1]];
        F       = Y*x_temp;
        Residu  = R- F;
        temp2   = QuanCast(Trans(delta)*Residu);
        counter = 0;
        while(temp2 > temp1*STOL + EPS) {
```

```
      counter = counter + 1;
      if(counter > MaxLineSearchCount) then {
         print " \n ========== \n Too many iterations for line search \n";
         break;
      } else {
         beta = beta/2.0;
         x_temp  = x + beta*delta;
         Y      = [2*x_temp[1][1], 1; 1, 2*x_temp[2][1]];
         F       = Y*x_temp;
         Residu  = R-F;
         temp2   = QuanCast(Trans(delta)*Residu);
     }
  }
  x = x_temp;

/* [c.2] : Restart for failed line search */

  if(counter > MaxLineSearchCount) then {
     ii = 1;
     print " \n **** Restart at new initial Value \n";
     Y          = [2*x[1][1], 1; 1, 2*x[2][1]];
     K          = [4*x[1][1], 1; 1, 4*x[2][1]];
     F          = Y*x;
     Residu     = R-F;
     K_inver    = Inverse(K);
     Residu_old  = Residu;
     delta_old   = Zero([dimen, 1]);
  } else {

  /* [c.3] : BFGS Update (i.e counter < MaxLineSearchCount) */

     gamma       = Residu_old - Residu;
     tem1        = QuanCast(Trans(delta)*Residu_old);
     tem1        = tem1*beta*beta;
     tem2        = QuanCast(Trans(delta)*gamma);
     tem2        = tem2*beta;
     ConditionNo = sqrt(tem2/tem1);
     V           = -ConditionNo*beta*Residu_old - gamma;
     W           = beta*delta/tem2;
     A           = I + V*Trans(W);
     K_inver     = Trans(A)*K_inver*A;

  /* [c.4] : Check convergence criteria */

     if (ConditionNo > 1E+5) {
         print"ConditionNo = ", ConditionNo, " \n";
         print"ERROR -- Condition Number Too Large \n";
         break;
     }

  /* [c.5] : Force and energy criteria */

     force_crt1 = L2Norm(Residu);
     force_crt2 = L2Norm(Residu0)*Ef;
```

```
          energy_crt1 = QuanCast(Trans(delta)*Residu_old);
          energy_crt1 = abs(energy_crt1);

          if (ii == 1) {
              energy_crt2 = QuanCast(Trans(delta)*Residu0);
              energy_crt2 = abs(energy_crt2*Ee);
          }

          if((force_crt1 <= force_crt2) && (energy_crt1 < energy_crt2)) {
              break;
          }

          Residu_old  = Residu;
          delta_old   = delta;
      }
   }

 /* [d] : Print results and terminate program execution */

   print" \n RESULTS : \n --------------\n";
   print" Iteration Number =", Iter_no, "\n";
   Y          = [2*x[1][1], 1; 1, 2*x[2][1]];
   F          = Y*x;
   PrintMatrix(x, F);
   quit;
```

The program output is

```
------initial Guess -----

MATRIX : "x"

row/col          1
      units
   1           7.00000e+00
   2           1.00000e+01

------Given R value -----

MATRIX : "R"

row/col          1
      units
   1           4.00000e+00
   2           5.00000e+00

 ==========
 Too many iterations for line search

 **** Restart at new initial Value
```

```
 RESULTS :
 --------------
 Iteration Number =    1.0000e+01

MATRIX : "x"

row/col          1
      units
   1           1.14237e+00
   2           1.38883e+00

MATRIX : "F"

row/col          1
      units
   1           3.99883e+00
   2           5.00004e+00
```

## 3.3 Han-Powell Algorithm for Optimization

In this section we will demonstrate how ALADDIN can be used to solve a series of optimization problems. Before we get to the details of the input file, we will briefly describe the Han-Powell algorithm for optimization.

Let $x$ be a n-dimensional vector, *f(x)* a n-dimensional function, and *g(x)* a m-dimensional function. The purpose of this section is to solve the mathematical problem

$$\begin{array}{ll} minimize & f(x) \\ subject\ to & g(x) = 0 \end{array} \tag{3.18}$$

by constructing a quadratic programming (QP) method, combined with an Armijo line search procedure, and Han-Powell's method of modified BFGS update (mathematical descriptions of these algorithms can be found in Luenberger [22]).

We begin by recalling that the Lagrange first-order necessary conditions for this problem amounts to solving the system of equations

$$\begin{array}{l} \nabla f(x) + \lambda^T \nabla g(x) = 0 \\ g(x) = 0 \end{array} \tag{3.19}$$

for $\mathbf{x}$ and $\lambda$. In terms of Lagrange function, above equation can be written as

$$\nabla l(x, \lambda) = 0$$

$$g(x) = 0$$

where $l(x, \lambda) = f(x) + g(x)\lambda$, and $\lambda$ is the Lagrange multiplier.

### 3.3.1 Quadratic Programming (QP)

The quadratic programming method is based on two mathematical assumptions, namely:

[1] Assume that the objective function *f(x)* may be approximated by a quadratic equation defined by a second order Taylor's series expansion about $x_k$.

[2] Assume that solutions to the constraint equation *g(x) = 0* are approximately given by a first order Taylors expansion about $x_k$.

Together, these assumptions imply

$$f(x_{k+1}) \approx f(x_k) + \nabla f(x_k)^T d + \frac{1}{2} d^T B_k d$$

$$g(x_{k+1}) \approx g(x_k) + \nabla g(x_k)^T d$$

The optimization problem described by equation (3.18) can be approximated by the following quadratic program problem

$$\begin{aligned} &minimize \;\; \nabla f(x_k)^T d + \tfrac{1}{2} d^T B_k d \\ &subject\ to \;\; g(x_k) + \nabla g(x_k)^T d = 0 \end{aligned} \tag{3.20}$$

where d $= x - x_k$, and $B_k$ is the Hessian of function f(x). Langrange's necessary conditions are

$$\begin{aligned} &B_k d + \nabla g(x_k)\lambda + \nabla f(x_k) = 0 \\ &g(x_k) + \nabla g(x_k)^T d = 0. \end{aligned} \tag{3.21}$$

and in this particular case, comprise an (n+m) dimensional linear system of equations. If $B_k$ is positive definite, and $\nabla g(x)$ has rank of m, then the matrix

$$\begin{bmatrix} B_k & \nabla g(x_k) \\ \nabla g(x_k)^T & 0 \end{bmatrix}$$

is nonsingular [22], and equations (3.21) may be solved with unique solutions

$$\begin{bmatrix} B_k & \nabla g(x_k) \\ \nabla g(x_k)^T & 0 \end{bmatrix} \begin{bmatrix} d \\ \lambda \end{bmatrix} = \begin{bmatrix} -\nabla f(x_k) \\ -g(x_k) \end{bmatrix} \tag{3.22}$$

to d and $\lambda$. The abovementioned QP solver gives the direction d for the next step. We will use the Armijo line search rule to determine how far an algorithm should move along this direction.

### 3.3.2 Armijo Line Search Rule

The essential idea is that the rule should first guarantee that the selected t is not too large. Let us define the function

$$\phi(t) = f(x_k + t\, d_k)$$

for fixed $\alpha$ and t = $(1, \beta, \beta^2, \beta^3, \cdots)$, a value of t is considered to be not too large if the corresponding function value lies below the dashed line; that is, if

$$\phi(t) \leq \phi(0) + \alpha\phi'(0)\, t \tag{3.23}$$

In this example, we chose $\alpha = 0.1, \beta = 0.5$.

### 3.3.3 The BFGS update and Han-Powell method

For the problem with equality constraints described in equation (3.19), the structured Newton method can be applied. We consider the iterative process defined by:

$$\begin{bmatrix} x_{k+1} \\ \lambda_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ \lambda_k \end{bmatrix} - \begin{bmatrix} B_k & \nabla g(x_k) \\ \nabla g(x_k)^T & 0 \end{bmatrix}^{-1} \begin{bmatrix} \nabla f(x_k, \lambda_k)^T \\ g(x_k) \end{bmatrix} \tag{3.24}$$

Let

$$y_k = \begin{bmatrix} x_k \\ \lambda \end{bmatrix} \quad A_k = \begin{bmatrix} B & \nabla g(x_k) \\ \nabla g(x_k)^T & 0 \end{bmatrix} \quad C_k = \begin{bmatrix} \nabla f(x_k, \lambda)^T \\ g(x_k) \end{bmatrix}$$

Notation for equation (3.24) can be simplified as:

$$y_{k+1} = y_k - A_k^{-1} C_k$$

The top part of system (3.22) can be rewritten as:

$$B_k d = -\nabla f(x_k)^T - \nabla g(x_k)\lambda = -\nabla l(x_k, \lambda) \tag{3.25}$$

and equation (3.25) can be implemented as a quasi-Newton method, with updates of matrix **B**

$$B_{k+1} = B_k + \frac{q_k q_k^T}{q_k^T p_k} - \frac{B_k p_k p_k^T B_k}{p_k^T B_k p_k}, \tag{3.26}$$

used to approximate the hessian of the lagrangian. In equation (3.26) $p_k = x_{k+1} - x_k$ and $q_k = \nabla l(x_{k+1}, \lambda_{k+1})^T - \nabla l(x_k, \lambda_{k+1})^T$. The Lagrange multiplier is calculated from solutions to (3.25). To ensure $B_k$ remains positive definite, the standard BFGS update is slightly altered.

**The Han-Powell Method:** The method consists of the following steps:

**Step 1 :** Start with an initial point $x_0$ and an initial positive definite matrix bf $B_0$ , set k = 0.

**Step 2 :** Solve the quadratic program described by system (3.20). If `d = 0` is a solution, then the current point satisfies the first order necessary conditions for a solution to the original problem (3.19).

**Step 3 :** With **d** found above, perform a line search in the direction **d**.

**Step 4 :** Update $B_k$, according to

$$
\begin{aligned}
B_{k+1} &= B_k - \frac{B_k p_k p_k^T B_k}{p_k^T B_k p_k} + \frac{r_k r_k^T}{p_k^T r_k} && (3.27)\\
p_k &= x_{k+1} - x_k && (3.28)\\
q_k &= \nabla l(x_{k+1}, \lambda_{k+1}) - \nabla l(x_k, \lambda_k) && (3.29)\\
r_k &= \theta_k q_k + (1-\theta_k) B_k p_k && (3.30)
\end{aligned}
$$

where $\lambda_{k+1}$ is the language multiplier vector of (3.20), and where

$$
\theta_k = \begin{cases} 1 & \text{if } p_k^T q_k \geq (0.2) p_k^T B_k p_k \\ \frac{(0.8) p_k^T B_k p_k}{p_k^T B_k p_k - p_k^T q_k} & \text{if } p_k^T q_k < (0.2) p_k^T B_k p_k. \end{cases} \tag{3.31}
$$

Again, detailed discussion on this method can be found in [22]

**Numerical Examples :** Consider the following three problems:

`Example 1` : Minimize a quadratic function with a set of linear constraints.

$$
\begin{aligned}
&\text{minimize}: \ f(x) = (x_1 - x_2)^2 + (x_2 + x_3 - 2)^2 + (x_4 - 1)^2 + (x_5 - 1)^2\\
&\text{subject to:} \ g_1(x) = x_1 + 3x_2 - 4 = 0\\
&\qquad\qquad\quad g_2(x) = x_3 + x_4 - 2x_5 = 0\\
&\qquad\qquad\quad g_3(x) = x_2 - x_5 = 0
\end{aligned} \tag{3.32}
$$

`Example 2` : Minimize a nonlinear non-quadratic function with a set of linear constraints.

$$
\begin{aligned}
&\text{minimize}: \ f(x) = (x_1 - 1)^2 + (x_1 - x_2)^2 + (x_2 - x_3)^4\\
&\text{subject to:} \ g_1(x) = x_1 + x_2 + 2x_3 - 4 = 0\\
&\qquad\qquad\quad g_2(x) = x_1 + x_3 - 2 = 0
\end{aligned} \tag{3.33}
$$

`Example 3` : Minimize a nonlinear non-quadratic function with nonlinear constraint.

$$
\begin{aligned}
&\text{minimize}: \ f(x) = (x_1 - 1)^2 + (x_1 - x_2)^2 + (x_2 - x_3)^4\\
&\text{subject to:} \ g(x) = x_1(1 + x_2^2) + x_3^4 - 3 = 0
\end{aligned} \tag{3.34}
$$

It is not difficult to observe that the analytical solution to the above examples are x = (1, 1, 1, 1, 1), x = (1,1,1) and x = (1, 1, 1).

Since the input commands for above examples are basically same except for the functions, we only illustrate one command file for example 1. But we give all the results for example 1, 2 and 3.

START OF INPUT FILE

```
/*
 *  ==================================================================
 *  Example Problem 1:
 *
 *    minmize f(x) = (x1 -x2)^2 + (x2+x3-2)^2 + (x4-1)^2 + (x5 -1)^2
 *
 *  Subject to constraints:
 *
 *         x1+3*x2            -4 = 0
 *                 x3+x4-2*x5    = 0
 *              x2        -x5    = 0
 *
 *  Notation -- f = f(x), G = g(x), Df = grad f(x), Dg = grad g(x).
 *
 *  Written By: X.G. Chen                                   June 1994
 *  ==================================================================
 */

SetUnitsOff;

/* [a] : Initialize and print variables and matrices */

   Epsilon = 1E-6;
   beta    = 0.5;
   alpha   = 0.1;
   IMAX    = 20; /* maximum of iterations */
   n       = 5;  /* number of variables   */
   M       = 3;  /* number of constraints */

   print "\n ****** Initial Guess Value ******* \n";

   x = [20; 0.1; 10; -1; -10];   /* not feasible */
   PrintMatrix(x);

   x_old      = x;
   lambda     = Zero([M, 1]);
   lambda_old = Zero([M, 1]);
   temp1 = (x[1][1] -x[2][1])*(x[1][1] -x[2][1]);
   temp2 = (x[2][1] +x[3][1] -2)*(x[2][1] +x[3][1] - 2);
   temp3 = (x[4][1] -1)*(x[4][1] -1);
   temp4 = (x[5][1] -1)*(x[5][1] -1);

   f = temp1 + temp2 + temp3 + temp4;
   A = Zero([n+M, n+M]);
   C = Zero([n+M, 1]);
   B  = Zero([n,n]);
   Df = Zero([n,1]);
   DF = Zero([n,1]);
   b  = Zero([M,1]);
   d  = Zero([n,1]);

   Df[1][1] =  2*(x[1][1]- x[2][1]);
   Df[2][1] = -2*(x[1][1]- x[2][1]) + 2*(x[2][1]+ x[3][1] -2);
   Df[3][1] =  2*(x[2][1]+ x[3][1]-2);
```

```
   Df[4][1] =  2*(x[4][1]- 1);
   Df[5][1] =  2*(x[5][1]- 1);

   B = Diag([n,1]); /* initialize B as identity matrix */

   b[1][1] = -4;
   b[2][1] =  0;
   b[3][1] =  0;

   k = 1;
   Trans_Dg = [1,3,0,0,0; 0,0,1,1,-2; 0, 1, 0, 0,-1];
   Dg       = Trans(Trans_Dg);
   G       = Trans_Dg*x+b;

/* [b] : Initialize matrices A and C */

   for ( i = 1; i <= n+M; i = i + 1) {
       for(j = 1; j <= n + M; j = j + 1) {
          if( i <= n) then {
             C[i][1] = -Df[i][1];
             if(j <= n) then {
                A[i][j] = B[i][j];
             } else {
                k = j - n;
                A[i][j] = Dg[i][k];
             }
          } else {
            k = i - n;
            C[i][1] = -G[k][1];
            if(j <= n) {
               A[i][j] = Trans_Dg[k][j];
            }
          }
       }
   }

   q = Zero([n,1]);

/* [c] : Main Loop */

   for ( ii = 1; ii <= IMAX; ii = ii + 1 ) {

   /* [c.1] : Solve QP for direction d */

      y = Solve(A,C);

      for ( i = 1; i <= n; i = i + 1) {
            d[i][1] = y[i][1];
      }

      for ( i = n+1; i <= n+M; i = i + 1) {
            j = i-n;
            lambda[j][1] = y[i][1];
      }
```

```
   if(L2Norm(d) <= Epsilon) {
      break;
   }

/* [c.2] : Line Search with Armijo's Rule */

   t       = 1;
   x       = x_old + t*d;
   temp1   = (x[1][1] -x[2][1])*(x[1][1] -x[2][1]);
   temp2   = (x[2][1] +x[3][1] -2)*(x[2][1] +x[3][1] - 2);
   temp3   = (x[4][1] -1)*(x[4][1] -1);
   temp4   = (x[5][1] -1)*(x[5][1] -1);
   f_new   = temp1 + temp2 + temp3 + temp4;

   temp    = alpha*QuanCast(Trans(Df)*d);
   counter = 0;
   while (f_new > f + t*temp) { /* t is too large */
      counter = counter + 1;
      t       = t*beta;
      x       = x_old + t*d;
      temp1   = (x[1][1] -x[2][1])*(x[1][1] -x[2][1]);
      temp2   = (x[2][1] +x[3][1] -2)*(x[2][1] +x[3][1] - 2);
      temp3   = (x[4][1] -1)*(x[4][1] -1);
      temp4   = (x[5][1] -1)*(x[5][1] -1);
      f_new   = temp1 + temp2 + temp3 + temp4;
      if(counter > 5) {
         print " Too much iterations for line search \n";
         print " counter =", counter," \n";
         break;
      }
   }

/* [c.3] : Modified BFGS matrix update */

   DF[1][1] =  2*(x[1][1]- x[2][1]);
   DF[2][1] = -2*(x[1][1]- x[2][1]) + 2*(x[2][1]+ x[3][1] -2);
   DF[3][1] =  2*(x[2][1]+ x[3][1]-2);
   DF[4][1] =  2*(x[4][1]- 1);
   DF[5][1] =  2*(x[5][1]- 1);

   q     = DF + Dg*lambda - Df - Dg*lambda_old;
   A1    = QuanCast(Trans(d)*B*d)*t*t;
   A2    = QuanCast(Trans(d)*q)*t;

   if(A2 >= (0.2*A1)) then {
      theta = 1.0;
   } else {
      theta = 0.8*A1/(A1-A2);
   }

   r          = theta*q + (1-theta)*t*B*d;
   A3         = QuanCast(Trans(d)*r)*t;
   B          = B - B*d*Trans(d)*B/A1 + r*Trans(r)/A3;
```

```
      Df          = DF;
      x_old       = x;
      lambda_old = lambda;
      G           = Trans_Dg*x+b;

      for ( i = 1; i <= n; i = i + 1) {
          C[i][1] = -Df[i][1];
          for(j = 1; j <= n; j = j + 1) {
              A[i][j] = B[i][j];
          }
      }

      for ( i = 1; i <= M; i = i + 1) {
            k = i + n;
            C[k][1] = -G[i][1];
      }
   }

/* [d] : Print results and terminate program execution */

   print" Results: \n --------------\n";
   print" Iteration Number =", ii-1, "\n\n";
   PrintMatrix(x);
   quit;
```

If B is initialized with identity matrix, then the results are:

**Example 1:**

```
 ****** Initial guess value *******

MATRIX : "x"

row/col          1
   1            2.00000e+01
   2            1.00000e-01
   3            1.00000e+01
   4           -1.00000e+00
   5           -1.00000e+01
 Results:
 --------------
 Iteration Number =    10

MATRIX : "x"

row/col          1
   1            1.00000e+00
   2            1.00000e+00
   3            1.00000e+00
   4            1.00000e+00
   5            1.00000e+00
```

**Example 2:**

```
 ****** Initial guess value *******

MATRIX : "x"

row/col          1
      units
   1            2.00000e+00
   2            2.00000e+00
   3            2.00000e+00
 Results:
 --------------
 Iteration Number =    5.0000e+00

MATRIX : "x"

row/col          1
      units
   1            9.99891e-01
   2            9.99891e-01
   3            1.00011e+00
```

**Example 3:** The process will not converge after maximum number of twenty iterations. And increase maximum number can not help the convergence.

```
****** Initial guess value *******

MATRIX : "x"

row/col          1
   1            2.00000e+00
   2            1.40000e+00
   3            1.50000e+00
 Results:
 --------------
 Iteration Number =    2.0000e+01

MATRIX : "x"

row/col          1
   1            1.00032e+00
   2            1.01113e+00
   3            9.94158e-01
```

If B is initialized with $B(x_0)$, then the results are:
**Example 1:**

```
 ****** Initial guess value *******

MATRIX : "x"
```

```
row/col          1
   1           2.00000e+01
   2           1.00000e-01
   3           1.00000e+01
   4          -1.00000e+00
   5          -1.00000e+01
 Results:
 --------------
 Iteration Number =    1.0000e+00

MATRIX : "x"

row/col          1
   1           1.00000e+00
   2           1.00000e+00
   3           1.00000e+00
   4           1.00000e+00
   5           1.00000e+00
```

### Example 2:

```
 ****** Initial guess value *******

MATRIX : "x"

row/col          1
      units
   1           2.00000e+00
   2           2.00000e+00
   3           2.00000e+00
 Results:
 --------------
 Iteration Number =    1.0000e+00

MATRIX : "x"

row/col          1
      units
   1           1.00000e+00
   2           1.00000e+00
   3           1.00000e+00
```

### Example 3:

```
 ****** Initial guess value *******

MATRIX : "x"

row/col          1
   1           2.00000e+00
   2           1.40000e+00
   3           1.50000e+00
 Results:
 --------------
```

```
 Iteration Number =    6.0000e+00

MATRIX : "x"

row/col          1
   1            9.99850e-01
   2            9.94974e-01
   3            1.00240e+00
```

In these three examples, the hessian of f(x) is stored in matrix `B`. The hessian is constant. Moreover, when the B matrix corresponds to the exact value, only one iteration of optimization is required to compute an optimal solution. The initial guess point for Example 2 is a feasible point. While the points for Example 1 and 3 are not feasible points.

# Chapter 4

# Computational Methods for Dynamic Analysis of Structures

## 4.1 Introduction

In this section we demonstrate how ALADDIN may be used to compute the time-history response of a linear elastic multi-degree of freedom system defined by the equation of equilibrium

$$\mathbf{M}\ddot{X}(t) + \mathbf{C}\dot{X}(t) + \mathbf{K}X(t) = \mathbf{P}(t). \tag{4.1}$$

In equation (4.1) $\mathbf{M}$, $\mathbf{C}$, and $\mathbf{K}$ are $(n \times n)$ mass, damping, and stiffness matrices, respectively. $\mathbf{P}(t)$, $\mathtt{X}(t)$, $\dot{\mathtt{X}}(t)$, and $\ddot{X}(t)$ are $(n \times 1)$ external load, displacement, velocity, and acceleration vectors at time `t`.

Two numerical procedures are described and demonstrated in this chapter. We begin with the method of Newmark Integration to compute the time-history response of a four story shear structure. Then we repeat the analysis using the method of Modal Analysis. In both cases, the shear structure is acted upon by a saw-toothed external force applied to the roof-level degree of freedom.

## 4.2 Method of Newmark Integration

Newmark integration methods [8] approximate the time dependent response of linear and nonlinear 2nd-order equations by insisting that equilibrium be satisfied only at a discrete number of points (or timesteps). If $(t)$ and $(t + \triangle t)$ are successive timesteps in the integration procedure, the two equations of equilibrium that must be satisfied are

$$\mathbf{M}\ddot{X}(t) + \mathbf{C}\dot{X}(t) + \mathbf{K}X(t) = \mathbf{P}(t), \tag{4.2}$$

and

$$\mathbf{M}\ddot{X}(t+\triangle t)+\mathbf{C}\dot{X}(t+\triangle t)+\mathbf{K}X(t+\triangle t)=\mathbf{P}(t+\triangle t). \tag{4.3}$$

Now let's assume that solutions to equation (4.2) are known, and (4.3) needs to be solved. At each timestep there are $3n$ unknowns corresponding to the displacement, velocity, and acceleration of each component of $X$. Since we only have $n$ equations, the natural relationship existing between the acceleration and velocity,

$$\dot{X}(t+\triangle t) \;=\dot{X}(t)+\textstyle\int_{(t)}^{(t+\triangle t)}\ddot{X}(\tau)d\tau, \tag{4.4}$$

and velocity and displacement

$$X(t+\triangle t) \;=X(t)+\textstyle\int_{(t)}^{(t+\triangle t)}\dot{X}(\tau)d\tau, \tag{4.5}$$

must be enforced to reduce the number of unknowns to $n$. $\ddot{X}(\tau)$ is an unknown function for the acceleration across the timestep. The Newmark family of integration methods assume that: (1) acceleration within the timestep behaves in a prescribed manner, and (2) the integral of acceleration across the timestep may be expressed as a linear combination of accelerations at the endpoints. Discrete counterparts to the continuous update in velocity and displacement are:

$$\dot{X}(t+\triangle t) \;=\; \dot{X}(t)+\triangle t\Big[(1-\gamma)\ddot{X}(t)+\gamma\ddot{X}(t+\triangle t)\Big] \tag{4.6}$$

$$X(t+\triangle t) \;=\; X(t)+\triangle t\dot{X}(t)+\frac{\triangle t^2}{2}\Big[(1-2\beta)\ddot{X}(t)+2\beta\ddot{X}(t+\triangle t)\Big] \tag{4.7}$$

with the parameters $\gamma$ and $\beta$ determining the accuracy and stability of the method under consideration. The equations for discrete update in velocity and displacement are substituted into equation (4.3) and rearranged to give:

$$\hat{\mathbf{M}}\ddot{X}(t+\triangle t)=\hat{\mathbf{P}}(t+\triangle t) \tag{4.8}$$

where

$$\hat{\mathbf{M}}=\mathbf{M}+\gamma\triangle t\mathbf{C}+\beta\triangle t^2\mathbf{K} \tag{4.9}$$

and

$$\begin{aligned}\hat{\mathbf{P}}(t+\triangle t) \;&=\mathbf{P}(t+\triangle t)-\mathbf{C}\dot{X}(t)-\mathbf{K}X(t)-\triangle t\,[\mathbf{K}]\,\dot{X}(t)-\\ &\qquad \triangle t\Big[(1-\gamma)\mathbf{C}+\tfrac{\triangle t}{2}(1-2\beta)\mathbf{K}\Big]\ddot{\mathbf{X}}(t).\end{aligned} \tag{4.10}$$

It is well known that when $\gamma=1/2$ and $\beta=1/4$, acceleration is constant within the timestep $t\in[t,t+\triangle t]$, and equal to the average of the endpoint accelerations. In such cases, approximations to the velocity and displacement will be linear and parabolic, respectively. Moreover, this discrete approximation is second order accurate and unconditionally stable. It conserves energy exactly for the free response vibration of linear undamped SDOF systems (we will check for conservation of energy in the numerical examples that follow).

**Numerical Example :** The method of Newmark Integration is demonstrated by computing the time-history response of the four story building structure caused by a time-varying external load applied to the roof level. Details of the shear building and external loading are shown in Figures 4.1 and 4.3.

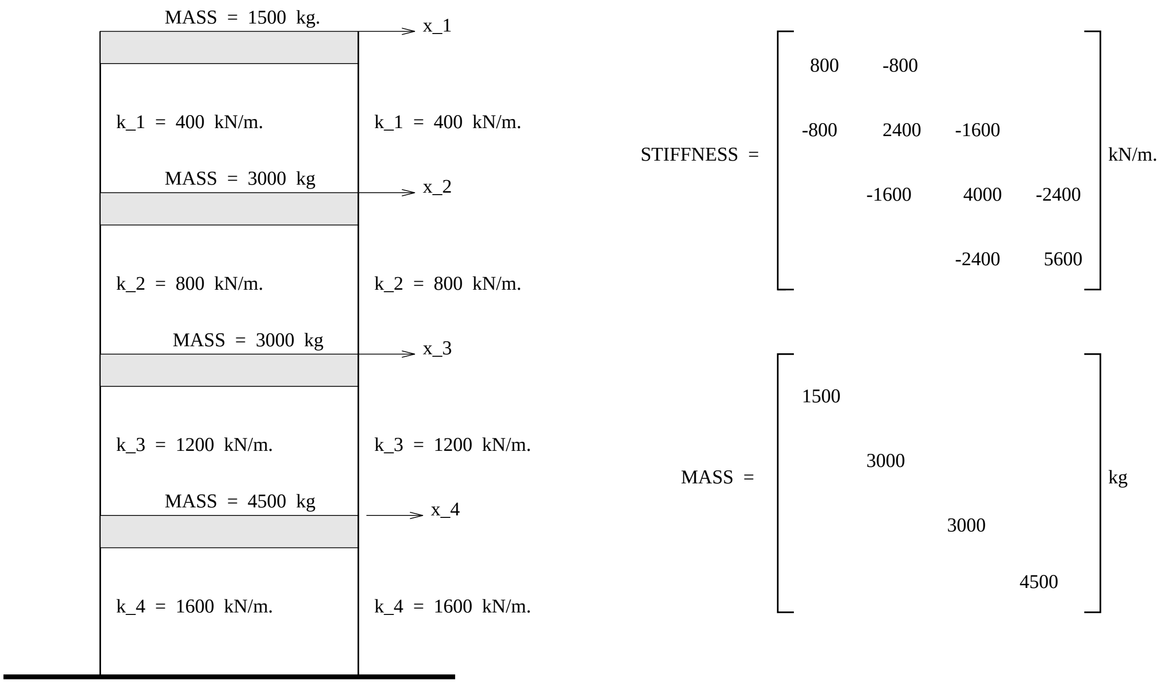


Figure 4.1: Schematic of Shear Building for Newmark Analysis

A simplified model of the building is obtained by assuming that all of the building mass is lumped at the floor levels, that the floor beams are rigid, and that the columns are axially rigid. Together these assumptions generate model that is commonly known as a shear-type building model, where displacements at each floor level may be described by one degree-of-freedom alone. Only four degrees of freedom are needed to describe total displacements of the structure.

Details of the mass and stiffness matrices are shown on the right-hand side of Figure 4.1. From a physical point of view, element `(i,j)` of the stiffness matrix corresponds to the nodal force that must be applied to degree of freedom `j` in order to produce a unit displacement at degree of freedom `i`, and zero displacements at all other degrees of freedom.

Figure 4.2 summarizes the mode shape, circular frequencies and natural period for each of the modes in the shear building. The shear building has a fundamental period of 0.5789 seconds.

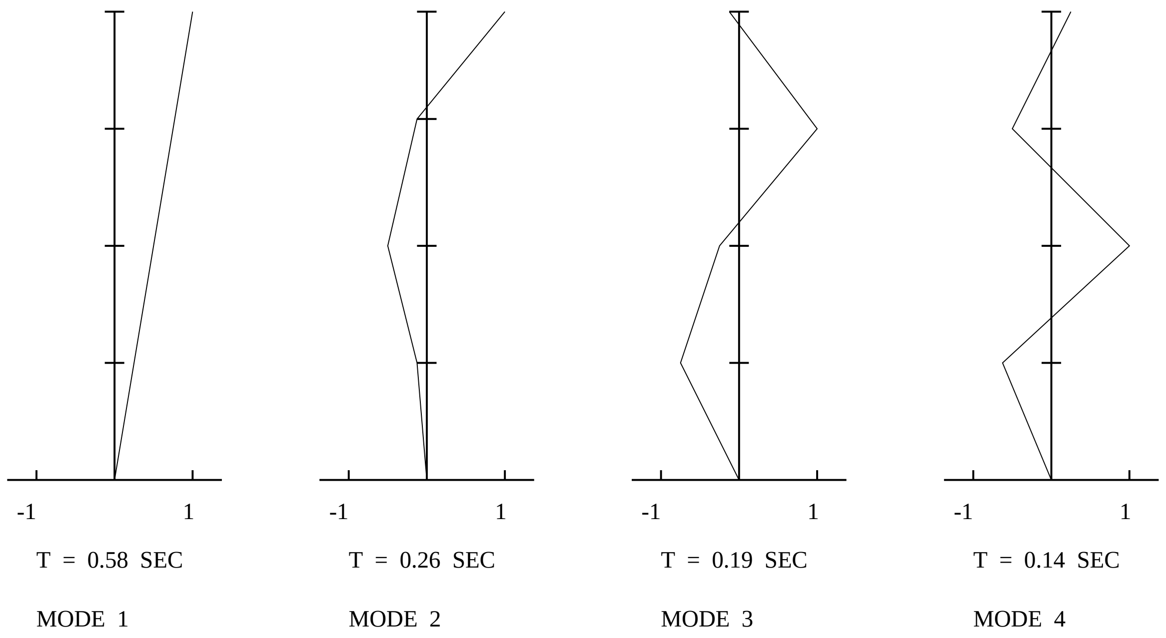


Figure 4.2: Mode Shapes and Natural Periods for Shear Building

The dynamic behavior of the shear building is generated by the external force, shown in Figure 4.3, applied to the roof-level degree of freedom.

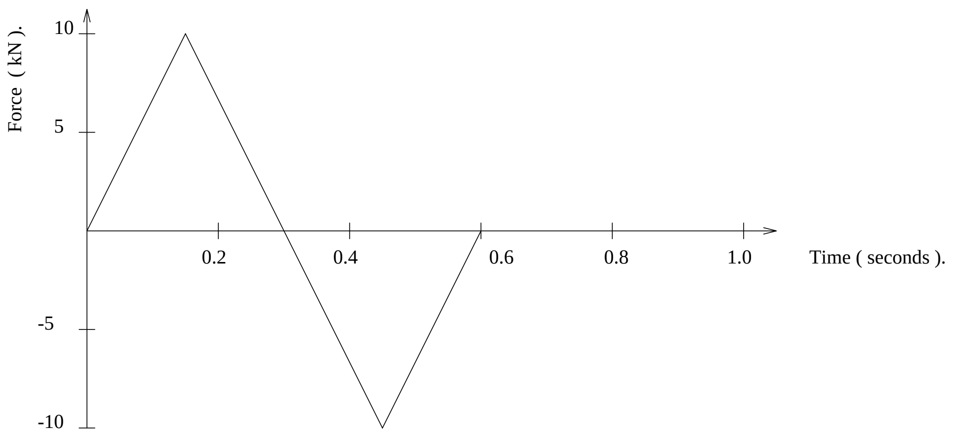


Figure 4.3: Externally Applied Force (kN) versus Time (sec)

Notice that we have selected the time-scale of the applied force so that it has a period close to the first natural period of the structure (i.e. 0.5789 seconds versus 0.6 seconds period for the applied load).

**Input File :** A seven-part input file is needed to define the mass and stiffness matrices, external loading, and solution procedure via the method of Newmark Integration. The step-by-step details of our Newmark Algorithm are:

[1] Form the stiffness matrix $\mathbf{K}$, the mass matrix $\mathbf{M}$, and the damping matrix $\mathbf{C}$. Compute the effective mass matrix $\hat{\mathbf{M}}$.

[2] Initialize the displacement $X(0)$ and velocity $\dot{X}(0)$ at time 0. Backsubstitute $X(0)$ and $\dot{X}(0)$ into equation (4.2), and solve for $\ddot{X}(0)$.

[3] Select an integration time step $\triangle t$, and Newmark parameters $\gamma$ and $\beta$.

[4] Enter Main Loop of Newmark Integration.

[5] Compute the effective load vector $\hat{\mathbf{P}}(t + \triangle t)$.

[6] Solve equation (4.8) for acceleration $\ddot{\mathbf{X}}(t + \triangle t)$.

[7] Compute $\dot{\mathbf{X}}(t + \triangle t)$ and $\mathbf{X}(t + \triangle t)$ by backsubstituting $\ddot{\mathbf{X}}(t + \triangle t)$ into the equations for discrete update in velocity and displacement.

[8] Go to Step [4].

Structural damping in the shear building is ignored.

**Input File :** The four story shear building, external loading, and newmark algorithm are defined in a seven-part input file.

START OF INPUT FILE

```
/* [a] : Parameters for Problem Size/Newmark Integration */

   dt     = 0.03 sec;
   nsteps =  200;
   gamma  = 0.50;
   beta   = 0.25;

/* [b] : Form Mass, stiffness and load matrices */

   mass = ColumnUnits( 1500*[ 1, 0, 0, 0;
                              0, 2, 0, 0;
                              0, 0, 2, 0;
                              0, 0, 0, 3], [kg] );

   stiff = ColumnUnits( 800*[ 1, -1,  0,  0;
                             -1,  3, -2,  0;
                              0, -2,  5, -3;
                              0,  0, -3,  7], [kN/m] );

   PrintMatrix(mass, stiff);
```

```
/*
 * [c] : Generate and print external saw-tooth external loading matrix. First and
 *       second columns contain time (sec), and external force (kN), respectively.
 */

   myload = ColumnUnits( Matrix([21,2]), [sec], [1]);
   myload = ColumnUnits( myload,          [kN], [2]);

   for(i = 1; i <= 6; i = i + 1) {
       myload[i][1] = (i-1)*dt;
       myload[i][2] = (2*i-2)*(1 kN);
   }

   for(i = 7; i <= 16; i = i + 1) {
       myload[i][1] = (i-1)*dt;
       myload[i][2] = (22-2*i)*(1 kN);
   }

   for(i = 17; i <= 21; i = i + 1) {
       myload[i][1] = (i-1)*dt;
       myload[i][2] = (2*i-42)*(1 kN);
   }

   PrintMatrix(myload);

/* [d] : Initialize working displacement, velocity, and acc'n vectors */

   displ  = ColumnUnits( Matrix([4,1]), [m]    );
   vel    = ColumnUnits( Matrix([4,1]), [m/sec]);
   accel  = ColumnUnits( Matrix([4,1]), [m/sec/sec]);
   eload  = ColumnUnits( Matrix([4,1]), [kN]);

/*
 * [e] : Allocate Matrix to store three response parameters -- Col 1 = time (sec);
 *       Col 2 = roof displacement (m); Col 3 = Total energy (N.m)
 */

   response = ColumnUnits( Matrix([nsteps+1,3]), [sec], [1]);
   response = ColumnUnits( response,  [cm], [2]);
   response = ColumnUnits( response, [Jou], [3]);

/* [f] : Compute (and compute LU decomposition) effective mass */

   MASS  = mass + stiff*beta*dt*dt;
   lu    = Decompose(Copy(MASS));

   for(i = 1; i <= nsteps; i = i + 1) {

    /* [f.1] : Update external load */

       if((i+1) <= 21) then {
          eload[1][1] = myload[i+1][2];
       } else {
          eload[1][1] = 0.0 kN;
```

```
      }

      R = eload - stiff*(displ + vel*dt + accel*(dt*dt/2.0)*(1-2*beta));

   /* [f.2] : Compute new acceleration, velocity and displacement  */

      accel_new = Substitution(lu,R);
      vel_new   = vel   + dt*(accel*(1.0-gamma) + gamma*accel_new);
      displ_new = displ + dt*vel + ((1 - 2*beta)*accel + 2*beta*accel_new)*dt*dt/2;

   /* [f.3] : Update and print new response */

      accel = accel_new;
      vel   = vel_new;
      displ = displ_new;

   /* [f.4] : Compute "kinetic + potential energy" */

      e1 = Trans(vel)*mass*vel;
      e2 = Trans(displ)*stiff*displ;
      energy = 0.5*(e1 + e2);

   /* [f.5] : Save components of time-history response */

      response[i+1][1] = i*dt;
      response[i+1][2] =  displ[1][1];
      response[i+1][3] = energy[1][1];
  }

/* [g] : Print response matrix and quit */

  PrintMatrix(response);
  quit;
```

Points to note are:

[1] In Part [a] of the input file, the variables `nstep` and `dt` are initialized for 200 timesteps of 0.03 seconds. The Newmark parameters $\gamma$ and $\beta$ (i.e. input file variables `gamma` and `beta`) are set to 0.5 and 0.25, respectively.

[2] Details of the external load versus time are generated for 0.6 seconds, and stored in an array "`myload`". Column one of "`myload`" contains time (seconds), and column two, the externally applied force (kN). The contents of "`myload`" are then transfered to the external load vector within the main loop of the Newmark Integration.

[3] Rather than print the results of the integration at the end of each timestep, we allocate memory for "`response`", a $(201 \times 3)$ matrix, and store relevant details of the computed response at the end of each iteration of the Newmark method. The first, second, and third columns of "`response`" contain the time (sec), displacement of the roof (cm), and total energy of the system (Joules).

[4] Theoretical considerations indicate – see note above – that after the external loading has finished, the sum of kinetic energy plus potential energy will be constant in the undamped shear structure. The total energy (Joules) is

$$\text{Total Energy} \;=\; \frac{1}{2}\left[\dot{X}(t)^T\mathbf{M}\dot{X}(t) + X(t)^T\mathbf{K}X(t)\right]. \tag{4.11}$$

[5] Matrix "`response`" is printed after the completion of the Newmark integration. This strategy of implementation has several benefits. First, the response quantities are easier to interpret if they are bundled together in one array. The second benefit occurs when program output is redirected to a file. Upon completion of the analysis, it is an easy process to edit the output file, and feed "`response`" into MATLAB. This is how we created Figures 4.4 to 4.9.

**Abbreviated Output File :** The output file contains summaries of the mass and stiffness matrices for the shear building, and abbreviated details of the external loading, "`myload`", and the response matrix "`response`".

START OF ABBREVIATED OUTPUT FILE

```
MATRIX : "mass"

row/col           1             2             3             4
      units          kg            kg            kg            kg
   1            1.50000e+03  0.00000e+00  0.00000e+00  0.00000e+00
   2            0.00000e+00  3.00000e+03  0.00000e+00  0.00000e+00
   3            0.00000e+00  0.00000e+00  3.00000e+03  0.00000e+00
   4            0.00000e+00  0.00000e+00  0.00000e+00  4.50000e+03

MATRIX : "stiff"

row/col           1             2             3             4
      units         N/m           N/m           N/m           N/m
   1            8.00000e+05 -8.00000e+05  0.00000e+00  0.00000e+00
   2           -8.00000e+05  2.40000e+06 -1.60000e+06  0.00000e+00
   3            0.00000e+00 -1.60000e+06  4.00000e+06 -2.40000e+06
   4            0.00000e+00  0.00000e+00 -2.40000e+06  5.60000e+06

MATRIX : "myload"

row/col           1             2
      units         sec           kN
   1            0.00000e+00  0.00000e+00
   2            3.00000e-02  2.00000e+00
   3            6.00000e-02  4.00000e+00

   .... details of "myload" matrix deleted ....

  19            5.40000e-01 -4.00000e+00
  20            5.70000e-01 -2.00000e+00
  21            6.00000e-01  0.00000e+00
```

```
MATRIX : "response"

row/col          1             2             3
      units          sec           cm           Jou
   1           0.00000e+00  0.00000e+00  0.00000e+00
   2           3.00000e-02  2.69339e-02  2.69339e-01
   3           6.00000e-02  1.51106e-01  3.99451e+00

  .... details of "response" matrix deleted ....

 199           5.94000e+00 -3.04363e+00  6.62976e+02
 200           5.97000e+00 -1.71841e+00  6.62976e+02
 201           6.00000e+00 -2.08155e-02  6.62976e+02
```

Points to note are:

[1] Figures 4.4 and 4.5 are time-history plots of the roof level displacement, and total system energy versus time. Given that the duration of external loading is only 0.6 seconds (see Figure 4.3), and that we have deliberately ignored structural damping, it is reassuring to see that total energy is conserved exactly beyond time = 0.6 seconds. You should also observe that the natural period of vibration in Figure 4.4 closely approximates 0.6 seconds, the lowest natural period of vibration in the structure.

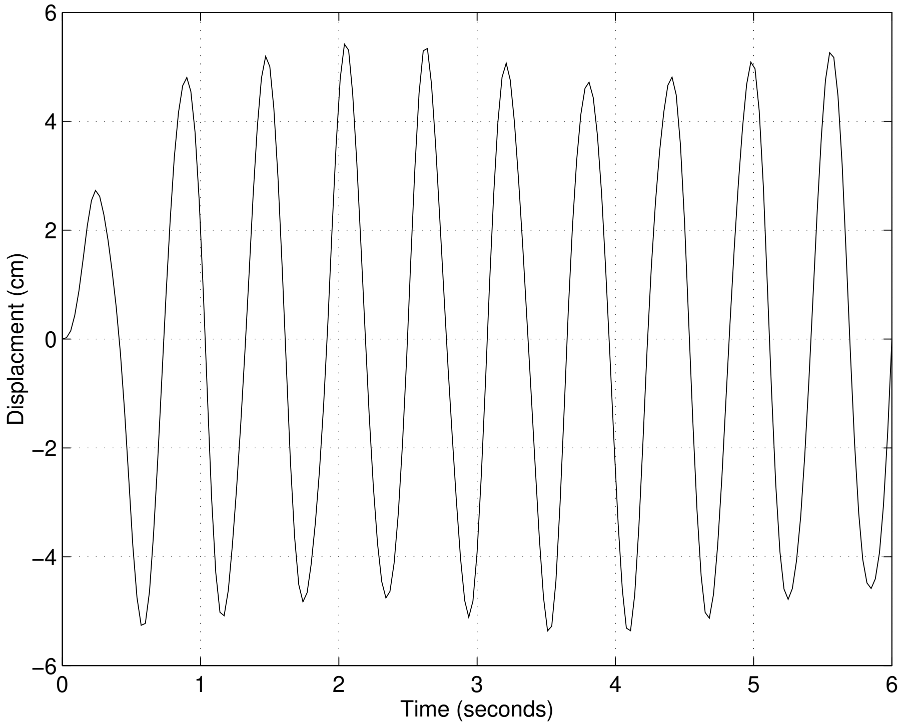


Figure 4.4: Newmark Integration : Lateral Displacement of Roof (cm) versus Time (sec)

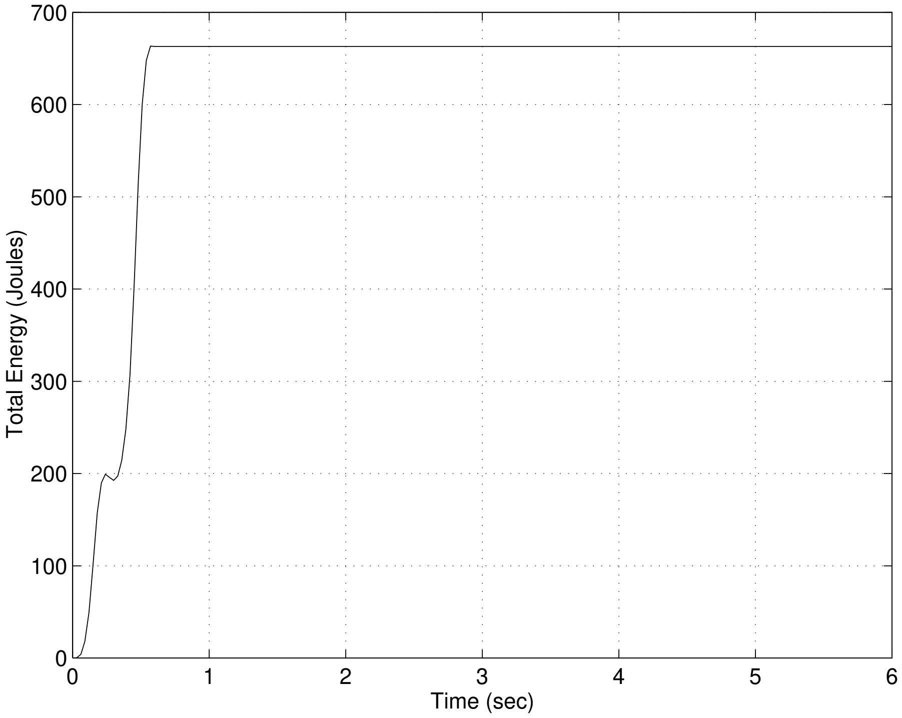


Figure 4.5: Newmark Integration : Total Energy (Joules) versus Time (sec)

## 4.3 Method of Modal Analysis

Let $\mathbf{\Phi}$ be a $(n \times p)$ nonsingular matrix $(p \leq n)$, and Y(t) be a $(p \times 1)$ matrix of time-varying generalized displacements. The objective of the method of modal analysis is to find a transformation

$$X(t) = \mathbf{\Phi} Y(t) \tag{4.12}$$

that will simplify the direct integration of equations (4.1). The modal equations are obtained by substituting equations (4.12) into (4.1), and then premultiplying (4.1) by $\Phi^T$. The result is:

$$\mathbf{\Phi^T M \Phi} \ddot{Y}(t) + \mathbf{\Phi^T C \Phi} \dot{Y}(t) + \mathbf{\Phi^T K \Phi} Y(t) = \mathbf{\Phi^T} P(t). \tag{4.13}$$

The notation in (4.13) may be simplified by defining the generalized mass matrix as $\mathbf{M}^* = \mathbf{\Phi^T M \Phi}$, the generalized damping matrix as $\mathbf{C}^* = \mathbf{\Phi^T C \Phi}$, the generalized stiffness matrix as $\mathbf{K}^* = \mathbf{\Phi^T K \Phi}$, and the generalized load matrix as $\mathbf{P}^*(t) = \mathbf{\Phi^T P}(t)$. Substituting these definitions into equation (4.13) gives

$$\mathbf{M}^* \ddot{Y}(t) + \mathbf{C}^* \dot{Y}(t) + \mathbf{K}^* Y(t) = \mathbf{P}^*(t). \tag{4.14}$$

The transformation $\mathbf{\Phi}$ is deemed to be effective if the bandwidth of matrices in (4.14) is much smaller than in equations (4.1). From a theoretical viewpoint, there may be many transformation matrices $\mathbf{\Phi}$ which will achieve this objective – a judicious choice of transformation matrix will work much better than many other transformation matrices, however.

To see how the method of modal analysis works in practice, consider the free vibration response of an undamped system

$$\mathbf{M} \ddot{X}(t) + \mathbf{K} X(t) = 0, \tag{4.15}$$

where $\mathbf{M}$ and $\mathbf{K}$ are $(n \times n)$ mass and stiffness matrices. We postulate that the time-history response of (4.15) may be approximated by a linear sum of $p$ harmonic solutions

$$X(t) = \sum_{i=1}^{p} \phi_i \cdot y_i(t) = \sum_{i=1}^{p} \phi_i \cdot A_i \cdot \sin(w_i t + \beta_i) = \mathbf{\Phi} \cdot Y(t), \tag{4.16}$$

where $y_i(t)$ is the ith component of $Y(t)$, and $\phi_i$ is the ith column of $\mathbf{\Phi}$. The amplitude and phase angle for the ith mode are given by $A_i$ and $\beta_i$, respectively – both quantities may be determined from the initial conditions of the motion. We solve the symmetric eigenvalue problem

$$\mathbf{K} \Phi = \mathbf{M} \Phi \Lambda \tag{4.17}$$

for $\mathbf{\Phi}$ and $\Lambda$ is a $(p \times p)$ diagonal matrix of eigenvalues

$$\Lambda = diag(\lambda_1, \lambda_2, \cdots, \lambda_p) = diag(w_1^2, w_2^2, \cdots, w_p^2). \tag{4.18}$$

It is well known that the eigenvectors of problem (4.16) will be orthogonal to both the mass and stiffness matrices. This means that the generalized mass and stiffness matrices will have zero terms except for diagonal terms. The generalized mass matrix takes the form

$$\mathbf{M}^* = \begin{bmatrix} {m_1}^* & 0 & \dots & 0 \\ 0 & {m_2}^* & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & {m_p}^* \end{bmatrix}, \tag{4.19}$$

and the generalized stiffness looks like

$$\mathbf{K}^* = \begin{bmatrix} {w_1}^2{m_1}^* & 0 & \dots & 0 \\ 0 & {w_2}^2{m_2}^* & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & {w_n}^2{m_p}^* \end{bmatrix}. \tag{4.20}$$

If the damping matrix, $\mathbf{C}$, is a linear combination of the mass and stiffness matrices, then the generalized damping matrix $\mathbf{C}^*$ will also be diagonal. A format that is very convenient for computation is

$$\mathbf{C}^* = \begin{bmatrix} 2\xi_1 w_1{m_1}^* & 0 & \dots & 0 \\ 0 & 2\xi_2 w_2{m_2}^* & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 2\xi_n w_n{m_p}^* \end{bmatrix}, \tag{4.21}$$

where $\xi_i$ is the ratio of critical damping for the ith mode of vibration.

For the undamped vibration of a linear multi-degree of freedom system, the eigenvalue/vector transformation is ideal because it reduces the bandwidth of $\mathbf{M}^*$, $\mathbf{C}^*$, and $\mathbf{K}^*$ to 1. In other words, the eigenvalue/vectors transform equation (4.1) from $n$ coupled equations into $p$ $(p \leq n)$ uncoupled single degree-of-freedom systems. The required computation is simplified because the total time-history response may now be evaluated in two (relatively simple) steps:

[1] Computation of the time-history responses for each of the $p$ single degree-of-freedom systems, followed by

[2] Combination of the SDOF's responses into the time-history response of the complete structure.

A number of computational methods are available to compute the time variation of displacements in each of the single degree of freedom systems. From a theoretical viewpoint, it can be shown that the total solution (or general solution) for a damped system is given by

$$y_i(t) = B_i(t)\sin(w_{id}t) + C_i(t)\cos(w_{id}t) \tag{4.22}$$

where $w_{id} = w_i\sqrt{1-\xi_i^2}$ is the damped circular frequency of vibration for the ith mode. The time-variation in coefficients $B_i(t)$ and $C_i(t)$ is given by

$$B_i(t) = e^{-\xi w_i t}\left[\frac{\dot{y}(0) + \xi w_i y(0)}{w_d} + \frac{1}{m_i^* w_{id}}\int_0^t P(\tau)e^{\xi_i w_i \tau}\cos(w_{id}\tau)d\tau\right] \tag{4.23}$$

and

$$C_i(t) = e^{-\xi_i w_i t}\left[y(0) - \frac{1}{m_i^* w_{id}}\int_0^t P(\tau)e^{\xi_i w_i \tau}\sin(w_{id}\tau)d\tau\right]. \tag{4.24}$$

where $y_i(0)$ and $\dot{y}_i(0)$ are the initial displacement and velocity for the ith mode.

If the details of $P(\tau)$ are simple enough, then analytic solutions to (4.22) may be possible. For most practical problems (e.g. earthquake ground motions), however, numerical solutions to $B_i(t)$ and $C_i(t)$ must be relied upon. A second approach, and the one that will be followed here, is to use Newmark Integration to compute the displacements for each of the individual modes.

**Numerical Problem :** We demonstrate the method of modal analysis by repeating the time-history computation defined in the previous section. Details of the shear building and external loading are shown in Figures 4.1 and 4.3.

**Input File :** The input file for modal analysis is a little more involved than the input file needed for Newmark Integration. This is due in part, to need to transform the equations of equilibrium to a simplier coordinate frame before the main loops of modal analysis begin.

In the input file that follows, $(2 \times 2)$ generalized mass and stiffness matrices are generated by first computing the $(4 \times 2)$ transformation matrix $\mathbf{\Phi}$ corresponding to the first two eigenvectors in the shear building. The elements of the generalized mass and stiffness matrices are zero, except for diagonal terms. In principle, each of the decoupled equations may be solved as single degree-of-freedom systems; in practice, however, it is computationally simplier to use the method of Newmark integration to solve both sets of decoupled equations together.

START OF INPUT FILE

```
/* [a] : Parameters for Modal Analysis and embedded Newmark Integration */

   no_eigen = 2;
```

```
   dt       = 0.03 sec;
   nsteps   = 200;
   beta     = 0.25;
   gamma    = 0.50;

/* [b] : Form Mass and stiffness matrices */

   mass = ColumnUnits( 1500*[ 1, 0, 0, 0;
                              0, 2, 0, 0;
                              0, 0, 2, 0;
                              0, 0, 0, 3], [kg] );

   stiff = ColumnUnits( 800*[ 1, -1,  0,  0;
                             -1,  3, -2,  0;
                              0, -2,  5, -3;
                              0,  0, -3,  7], [kN/m] );

   PrintMatrix(mass, stiff);

/* [c] : First two eigenvalues, periods, and eigenvectors */

   eigen       = Eigen(stiff, mass, [no_eigen]);
   eigenvalue  = Eigenvalue(eigen);
   eigenvector = Eigenvector(eigen);

   for(i = 1; i <= no_eigen; i = i + 1) {
       print "Mode", i ," : w^2 = ", eigenvalue[i][1];
       print " : T = ", 2*PI/sqrt(eigenvalue[i][1]) ,"\n";
   }

   PrintMatrix(eigenvector);

/* [d] : Generalized mass and stiffness matrices */

   EigenTrans = Trans(eigenvector);
   Mstar   = EigenTrans*mass*eigenvector;
   Kstar   = EigenTrans*stiff*eigenvector;

   PrintMatrix( Mstar );
   PrintMatrix( Kstar );

/*
 * [e] : Generate and print external saw-tooth external loading matrix. First and
 *       second columns contain time (sec), and external force (kN), respectively.
 */

   myload = ColumnUnits( Matrix([21,2]), [sec], [1]);
   myload = ColumnUnits( myload,          [kN], [2]);

   for(i = 1; i <= 6; i = i + 1) {
       myload[i][1] = (i-1)*dt;
       myload[i][2] = (2*i-2)*(1 kN);
   }
```

```
   for(i = 7; i <= 16; i = i + 1) {
       myload[i][1] = (i-1)*dt;
       myload[i][2] = (22-2*i)*(1 kN);
   }

   for(i = 17; i <= 21; i = i + 1) {
       myload[i][1] = (i-1)*dt;
       myload[i][2] = (2*i-42)*(1 kN);
   }

   PrintMatrix(myload);

/* [f] : Initialize system displacement, velocity, and load vectors */

   displ  = ColumnUnits( Matrix([4,1]), [m]    );
   vel    = ColumnUnits( Matrix([4,1]), [m/sec]);
   eload  = ColumnUnits( Matrix([4,1]), [kN]);

/* [g] : Initialize modal displacement, velocity, and acc'n vectors */

   Mdispl  = ColumnUnits( Matrix([ no_eigen,1 ]), [m]    );
   Mvel    = ColumnUnits( Matrix([ no_eigen,1 ]), [m/sec]);
   Maccel  = ColumnUnits( Matrix([ no_eigen,1 ]), [m/sec/sec]);

/*
 * [g] : Allocate Matrix to store five response parameters --
 *       Col 1 = time (sec);
 *       Col 2 = 1st mode displacement (cm);
 *       Col 3 = 2nd mode displacement (cm);
 *       Col 4 = 1st + 2nd mode displacement (cm);
 *       Col 5 = Total energy (Joules)
 */

   response = ColumnUnits( Matrix([nsteps+1,5]), [sec], [1]);
   response = ColumnUnits( response,  [cm], [2]);
   response = ColumnUnits( response,  [cm], [3]);
   response = ColumnUnits( response,  [cm], [4]);
   response = ColumnUnits( response, [Jou], [5]);

/* [h] : Compute (and compute LU decomposition) effective mass */

   MASS  = Mstar + Kstar*beta*dt*dt;
   lu    = Decompose(MASS);

/* [i] : Mode-Displacement Solution for Response of Undamped MDOF System  */

   for(i = 1; i <= nsteps; i = i + 1) {

    /* [i.1] : Update external load */

       if((i+1) <= 21) then {
          eload[1][1] = myload[i+1][2];
       } else {
          eload[1][1] = 0.0 kN;
```

```
        }

        Pstar = EigenTrans*eload;
        R = Pstar - Kstar*(Mdispl + Mvel*dt + Maccel*(dt*dt/2.0)*(1-2*beta));

     /* [i.2] : Compute new acceleration, velocity and displacement  */

        Maccel_new = Substitution(lu,R);
        Mvel_new   = Mvel   + dt*(Maccel*(1.0-gamma) + gamma*Maccel_new);
        Mdispl_new = Mdispl + dt*Mvel + ((1 - 2*beta)*Maccel + 2*beta*Maccel_new)*dt*dt/2;

     /* [i.3] : Update and print new response */

        Maccel = Maccel_new;
        Mvel   = Mvel_new;
        Mdispl = Mdispl_new;

     /* [i.4] : Combine Modes */

        displ = eigenvector*Mdispl;
        vel   = eigenvector*Mvel;

     /* [i.5] : Compute Total System Energy */

        e1 = Trans(vel)*mass*vel;
        e2 = Trans(displ)*stiff*displ;
        energy = 0.5*(e1 + e2);

     /* [i.6] : Save components of time-history response */

        response[i+1][1] = i*dt;                            /* Time                  */
        response[i+1][2] = eigenvector[1][1]*Mdispl[1][1];  /* 1st mode displacement */
        response[i+1][3] = eigenvector[1][2]*Mdispl[2][1];  /* 2nd mode displacement */
        response[i+1][4] = displ[1][1];               /* 1st + 2nd mode displacement */
        response[i+1][5] = energy[1][1];                    /* System Energy         */
    }

/* [j] : Print response matrix and quit */

    PrintMatrix(response);
    quit;
```

Points to note are:

[1] In Part [a] of the input file, the variables `nstep` and `dt` are initialized for 200 timesteps of 0.03 seconds. The Newmark parameters $\gamma$ and $\beta$ (i.e. input file variables `gamma` and `beta`) are set to 0.5 and 0.25, respectively.

[2] Overall behavior of the system is represented by $(4 \times 4)$ global mass and stiffness matrices. A $(4 \times 2)$ transformation matrix, $\Phi$, is computed by solving equation

(4.17) for the first two eigenvectors. As a result, we expect that the generalized mass and stiffness will be $(2 \times 2)$ diagonal matrices.

[3] The main loop of our modal analysis employs Newmark integration to compute the time-history response for the two decoupled equations.

**Abbreviated Output File :** The abbreviated output file contains only the essential details of output generated by the modal analysis. These details include the first two mode shapes of the shear building, the generalized mass and stiffness matrices, and heavily edited versions of matrices "`myload`" and "`response`".

START OF ABBREVIATED OUTPUT FILE

```
MATRIX : "mass"

   .... see Newmark output for details of "mass" ....

MATRIX : "stiff"

   .... see Newmark output for details of "stiff" ....

*** SUBSPACE ITERATION CONVERGED IN 11 ITERATIONS

Mode    1.0000e+00  : w^2 =    117.8 rad.sec^-2.0 : T =   0.5789 sec
Mode    2.0000e+00  : w^2 =    586.5 rad.sec^-2.0 : T =   0.2595 sec

MATRIX : "eigenvector"

row/col          1            2
      units
   1           1.00000e+00  1.00000e+00
   2           7.79103e-01 -1.00304e-01
   3           4.96553e-01 -5.39405e-01
   4           2.35062e-01 -4.36791e-01

MATRIX : "Mstar"

row/col          1            2
      units           kg           kg
   1           4.30934e+03  5.68434e-14
   2           1.70530e-13  3.26160e+03

MATRIX : "Kstar"

row/col          1            2
      units          N/m          N/m
   1           5.07691e+05 -1.96451e-10
   2          -2.91038e-10  1.91282e+06

MATRIX : "myload"

row/col          1            2
```

```
      units          sec           kN
   1            0.00000e+00  0.00000e+00
   2            3.00000e-02  2.00000e+00

   ...... details of external load removed ....

  20            5.70000e-01 -2.00000e+00
  21            6.00000e-01  0.00000e+00

MATRIX : "response"

row/col          1            2            3            4            5
      units          sec           cm           cm           cm          Jou
   1            0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00
   2            3.00000e-02  1.01728e-02  1.21886e-02  2.23614e-02  2.23614e-01

   ...... details of response matrix removed ....

 200            5.97000e+00 -1.26509e+00 -3.65074e-01 -1.63016e+00  6.59154e+02
 201            6.00000e+00  3.48910e-01 -3.36893e-01  1.20176e-02  6.59154e+02
```

Points to note are:

[1] Figures 4.6 and 4.7 show the time-history response for the first and second modes, respectively. Notice that the amplitude of vibration for the first mode is an order of magnitude larger than for the second mode. You should also observe that after the external load finishes at time = 0.6 seconds, the amplitude of vibration is constant within each mode, with the natural periods of vibration closely matching the eigenvalues/periods shown in Figure 4.2.

[2] The combined first + second modal response versus time is shown in Figure 4.8. It time-displacement curve is virtually identical to that computed with the method of Newmark integration.

[3] The time variation in total energy versus time is shown in Figure 4.9. One benefit of embedding Newmark inside the modal equations is that energy is conserved for each if the modes. For the integration beyond time = 0.6 seconds, the total energy is conserved at 659 Joules. This quantity compares to 662 Joules for the complete Newmark computation.

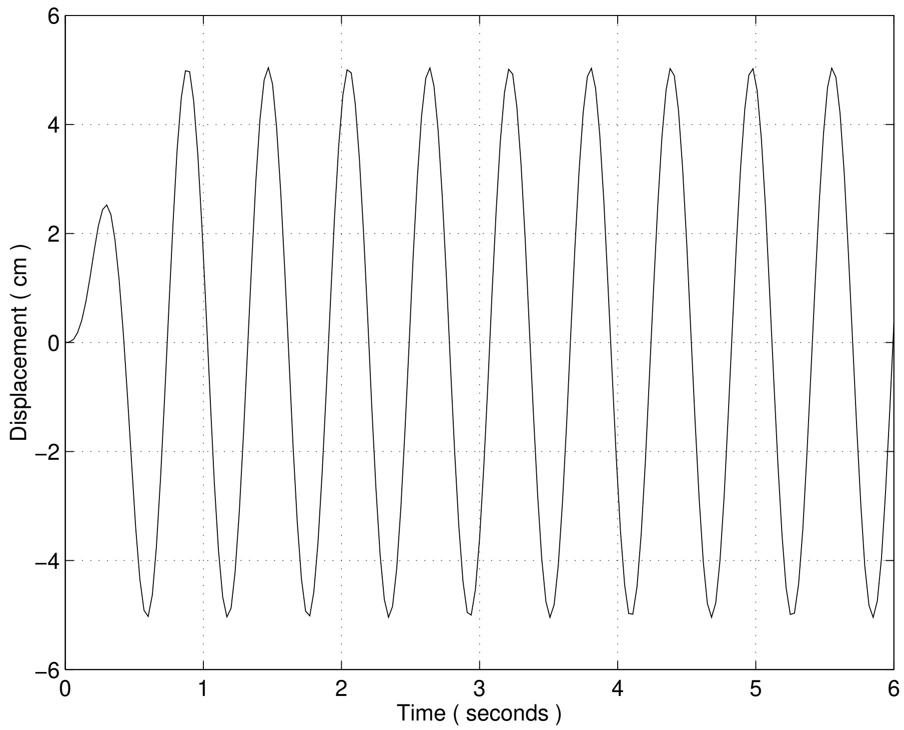


Figure 4.6: Modal Analysis : First Mode Displacement of Roof (cm) versus Time (sec)

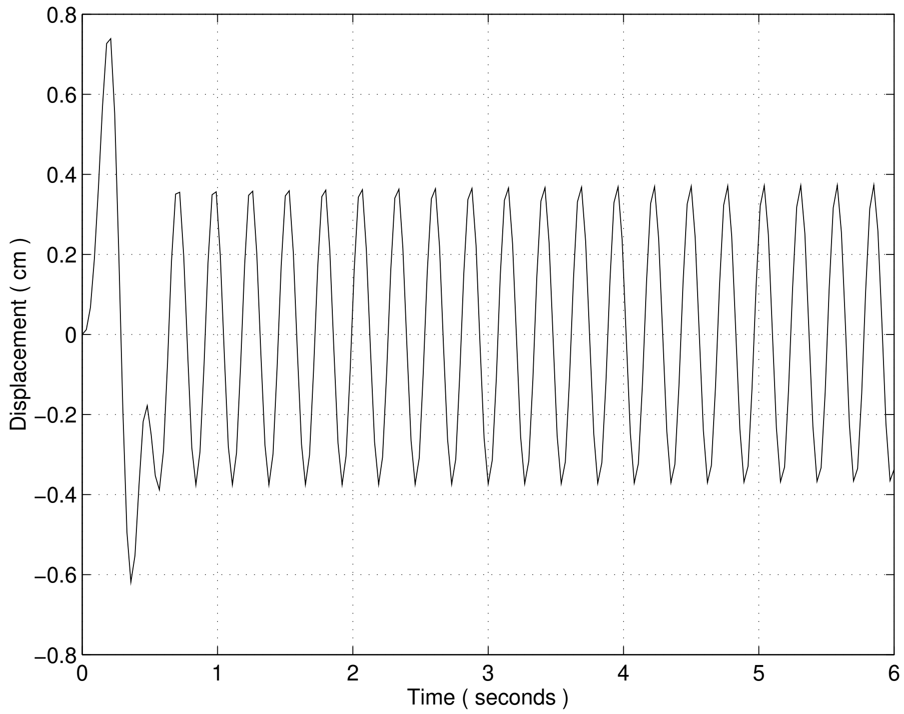


Figure 4.7: Modal Analysis : Second Mode Displacement of Roof (cm) versus Time (sec)

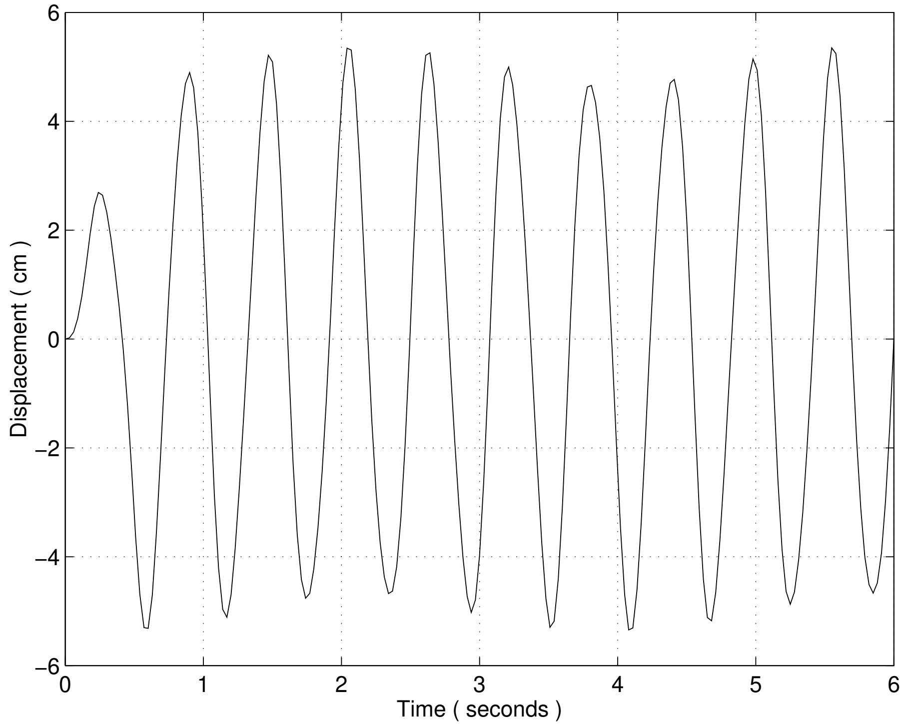


Figure 4.8: Modal Analysis : First + Second Mode Displacement of Roof (cm) versus Time (sec)

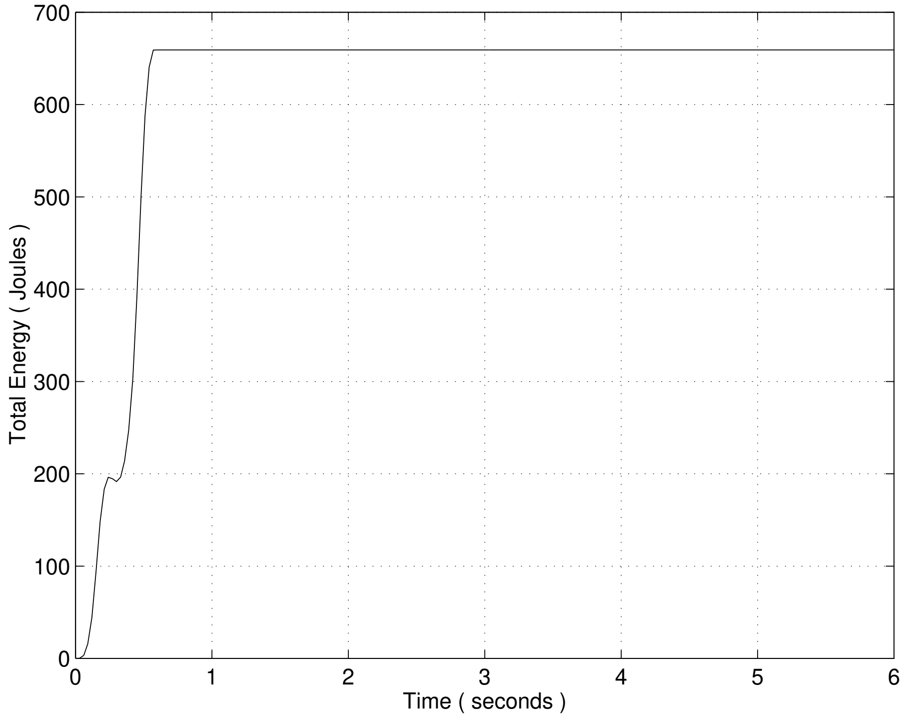


Figure 4.9: Modal Analysis : Total Energy (Joules) versus Time (sec)

# Part III

# FINITE ELEMENT LIBRARY

# Chapter 5

# Finite Element Analysis Language

## 5.1 Introduction

In this chapter we describe ALADDIN's built-in functions for the solution of finite element problems. Built-in function are provided for: (a) the generation of finite element meshes, (b) the definition of external loads, (c) the specification of boundary conditions, (d) the specification of section and material properties, and (e) linking finite element degrees of freedom.

With the finite element mesh in place, we can then compute solutions to a particular linear/nonlinear finite elment problem using the algorithms described in the previous chapters. To assist the engineer with basic finite element computations, built-in functions are also provided for the assembly of global stiffness matrices, global mass matrices, and external load vectors. An engineer may query the finite element database for information on the finite element mesh, section and material properties, and on the computed displacements and stresses. Together, these facilities provide an engineer with the means to write ALADDIN input files that will compare the performance of a structure against families of design rules.

## 5.2 Structure of Finite Element Input Files

The following diagram is a schematic of the main components in an ALADDIN input file suitable for the specification and solution of a finite element problem.

START OF INPUT FILE

```
/*
 *  =======================================================
 *  A description of the finite element problem goes here...
 *  =======================================================
 */
```

```
  ... Part [1] : Problem specification parameters.

StartMesh();

  ... Part [2] : Generate finite element mesh. Specify section and material
                 properties, external loads, and boundary conditions.

EndMesh();

  ... Part [3] : Describe solution procedure for finite element problem.

  ... Part [4] : If applicable, check performance of structure against
                 design rules.

  ... Part [5] : If applicable, generate arrays of output that are suitable
                 for plotting with MATLAB.

quit;
```

Every ALADDIN input file should begin with a description of the file's purpose, and who wrote it.

The problem specification parameters allow an engineer to state whether a finite element problem will be two- or three- dimensional, the maximum number of degree of freedom per node, and the maximum number of nodes per element. With this information in place, `StartMesh()` allocates the working memory for the finite element data structures.

ALADDIN statements are written for the finite element mesh generation, for the section and material properties, the specification of external loads, and for the boundary conditions in Part [2] of the input file. Often the overall size of a finite element model can be reduced by linking degrees of freedom – functions to link nodal degrees of freedom are also specified in Part [2]. `Endmesh()` loads the information provided in Part [2] into the finite element database.

Part [3] usually begins with the assembly of the global stiffness matrix, an external vectors, and if applicable, assembly of a global mass matrix. The algorithms presented in the previous chapters (e.g. newmark integration; modal analysis; optimization procedures) can be inserted here to solve a specific linear/nonlinear static/dynamic finite element problem.

In Parts [4] and [5] ALADDIN statements are written for design rule checking, and for the output of arrays that are suitable for plotting with MATLAB. ALADDIN input file should be terminated with the command `quit`.

## 5.3 Problem Specification Parameters

ALADDIN's specification parameters are:

- `NDimension` is short for "number of dimensions." For two- and three-dimensional finite element analyses, `NDimension` equals 2 and 3, respectively.
- `NDofPerNode` is short for Maximum number of degrees of freedom per nodal coordinate. When `NDimension` equals 2, `NDofPerNode` will take the value 3. And when `NDimension` equals 3, `NDofPerNode` will equal 6.
- `MaxNodesPerElement` corresponds to the maximum number of nodes per finite element.
- `InPlaneIntegPts` corresponds to the number of in-plane integration points for shell finite elements.
- `ThicknessIntegPts` number of layers of integration points through thickness direction of shell finite elements.

These parameter settings are used in the allocation of memory for the finite element mesh.

**Short Example :** In this short example, we initialize the problem specification parameters for a three-dimensional analysis of a structure with an eight-node shell finite element.

ABBREVIATED INPUT FILE

```
/* [a] : Define Problem Specific Parameters

   NDimension         = 3;
   NDofPerNode        = 5;
   MaxNodesPerElement = 8;   /* .... etc ..... */
```

## 5.4 Adding Nodes and Finite Elements

Two functions, `AddNode()` and `AddElement()`, are employed for the generation of finite element nodal coordinates, and the attachment of finite elements to the nodes.

- `AddNode( nodeno , coord_vector);` : Here `nodeno` is the node number in the finite element mesh. For two-dimensional finite element problems, `coord_vector` is a $(1 \times 2)$ matrix containing the [x, y] nodal coordinates. When the finite element problem is three-dimensional, `coord_vector` is a $(1 \times 3)$ matrix containing the [x, y, z] nodal coordinates.

- `AddElmt( elmtno , connect_vector, "name_of_elmt_attr");` : Here `elmtno` is the element number in the finite element mesh. `connect_vector` is a $(1 \times n)$ matrix containing a list of $n$ nodes to which the finite element will be attached. `"name_of_elmt_attr"` is the attribute name representing the finite element's material and section properties – for details, see Section 5.5.

**Short Example :** Figure 5.1 shows a two-dimensional coordinate system, and a line of six finite element nodes connected by 2-node beam finite elements. The nodes are located at y-coordinate = 1 $m$, and are spaced along the x-axis at 1 $m$ centers, beginning at x = 1 $m$ and finishing at x = 6 $m$.

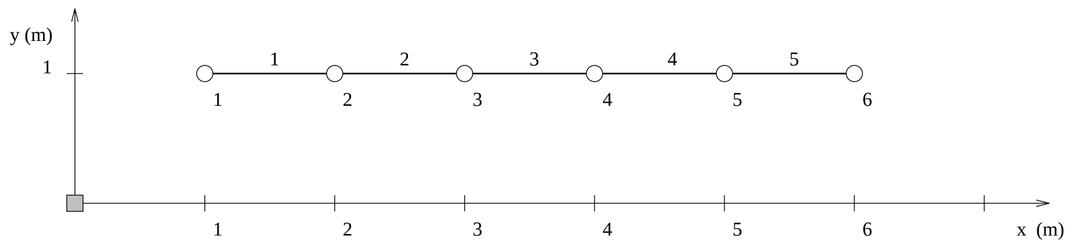


Figure 5.1: Line of Nodal Coordinates and Beam Finite Elements

ALADDIN's looping constructs are ideally suited for specification of the finite element nodes in a compact manner, and for the attachment of the two-node finite elements.

ABBREVIATED INPUT FILE

```
print "*** Generate grid of nodes for finite element model \n\n";

nodeno = 0;
x = 1 m; y = 1 m;
while(x <= 6 m) {
    nodeno = nodeno + 1;
    AddNode(nodeno, [x, y]);
    x = x + 1 m;
}

print "*** Attach finite elements to nodes \n\n";

elmtno = 0; nodeno = 0;
while(elmtno < 5) {
    elmtno = elmtno + 1; nodeno = nodeno + 1;
    AddElmt( elmtno, [nodeno, nodeno + 1], "name_of_elmt_attr");
}
```

In the first half of the script, 6 nodal coordinates are added to ALADDIN's database. Notice how, in the second block of code, we have used the notation `[nodeno, nodeno + 1]` to generate $(1 \times 2)$ matrices containing the node numbers that `elmtno` will be attached to.

## 5.5 Material and Section Properties

The attributes of element, section, and material types are specified with three functions, `ElementAttr()`, `SectionAttr()` and `MaterialAttr()`, followed by parameters inserted between braces $\{\cdots\}$. The details of each function are:

- `ElementAttr("name_of_element_attr")` $\{\cdots\}$;. Here `"name_of_elmt_attr"` is a character string for the name of the element attribute used in function calls to `AddElmt(···)`. Three character string arguments should be specified between the braces.

  `type` is a name for the finite element type (e.g. FRAME_2D for two dimensional frame elements). A list of finite element types may be found in Section 5.11.

  Character strings for the `section` and `material` attributes may taken from the section file `section.h` and the material file `material.h`, or may be user-defined with `SectionAttr("...")` and `MaterialAttr("...")`.

- `SectionAttr("name_of_section_attr")` $\{\cdots\}$;. Here `"name_of_section_attr"` is a character string for the name of the section attribute. Table 5.1 contains a list of section property names, their meaning, and the units associated with each property.

- `MaterialAttr("name_of_material_attr")` $\{\cdots\}$;. Here `"name_of_material_attr"` is a character string for the name of the material attribute. Table 5.2 contains a list of material property names, their meaning, and the units associated with each property.

When the number of (local) degrees of freedom in a finite element node is fewer than the number of (global) degrees of freedom at the node to which it is attached, a mapping must be provided from the local to global d.o.f. The syntax for such a mapping is

```
map ldof [A] to gdof [B].
```

where `[A]` and `[B]` are $(1 \times n)$ matrices. We will see, for example, that when plate finite elements (`type = PLATE`) are employed in a three-dimensional mesh, a suitable mapping is `map ldof [1,2,3] to gdof [2,4,6]`. This specification is provided as a fourth entry to `ElementAttr()`.

**Short Example :** The following script loads a finite element attribute called "floorelmts" into ALADDIN's database. The "floorelmts" attribute has three components – the finite element type is set to `FRAME_2D`, for two-dimensional beam column finite elements. The element's section and material properties are defined via links to the section attribute "floorsection," and the material attribute "floormaterial."

ABBREVIATED INPUT FILE

```
print "*** Define element, section, and material properties \n\n";

ElementAttr("floorelmts") { type     = "FRAME_2D";
                            section  = "mysection";
                            material = "ELASTIC";
                          }

SectionAttr("floorsection") { Ixy     = 1 m^4;
                              Iyy     = 2 m^4;
                              Ixx     = 3 m^4;
                              Izz     = 0.66666667 m^4;
                              depth   =   2 m;
                              width   =   1.5 m;
                            }

MaterialAttr("floormaterial") {  E    = 1E+7 kN/m^2;
                                 density = 0.1024E-5 kg/m^3;
                                 poisson = 1.0/3.0;
                                 yield   = 36000 psi;
                              }
```

---

You should notice that the section and material properties components are supplied, as required by the `FRAME_2D` element specifications described in Section 5.11.

Two functions, `GetSection()` and `GetMaterial()`, are provided for the retrieval of section and material properties. `GetSection([elmtno])` takes as its argument, a $(1 \times 1)$ matrix containing the element no. It returns a $16 \times 1$ matrix of section properties:

```
===================================================================
Matrix Element      Quantity     Description
===================================================================
       [1][1]            Ixx     Moment of inertia about x-x axis.
       [2][1]            Iyy     Moment of inertia about y-y axis.
       [3][1]            Izz     Moment of inertia about z-z axis.
       [4][1]            Ixz     Product of inertia x-z.
       [5][1]            Ixy     Product of inertia x-y.
       [6][1]            Iyz     Product of inertia y-z.
       [7][1]         weight     Section weight
       [8][1]             bf     Width of flange
       [9][1]             tf     Thickness of flange
      [10][1]          depth     Section depth
      [11][1]           area     Section area
      [12][1]      thickness     Thickness of plate
      [13][1]      tor_const     Torsional Constant J
      [14][1]             rT     Section radius of gyration
      [15][1]          width     Section width
      [16][1]             tw     Thickness of web
```

Similarly, `GetMaterial([elmtno])` takes as its argument, a $(1 \times 1)$ matrix containing the element no. It returns a $10 \times 1$ matrix of material properties:

| Name | Description | Units ($M^{\alpha}.L^{\beta}.T^{\gamma}$) |
|---|---|---|
| area | Area of cross section. | $L^2$ |
| depth | Depth of section. | $L$ |
| thickness | Thickness of section. | $L$ |
| Ixx | Moment of inertia about x-x axis. | $L^4$ |
| Iyy | Moment of inertia about y-y axis. | $L^4$ |
| Izz | Moment of inertia about z-z axis. | $L^4$ |
| Ixy and Iyx | Moment of inertia about x-y axes. | $L^4$ |
| Ixz and Izx | Moment of inertia about x-z axes. | $L^4$ |
| Iyz and Izy | Moment of inertia about y-z axes. | $L^4$ |
| rT | Radius of Gyration. | $L$ |
| tw | Width of web. | $L$ |
| tf | Width of flange. | $L$ |
| J | Torsional Constant. | $L^4$ |
| width | Width of section. | $L$ |

Table 5.1: **Section Properties**

```
=========================================================================
Matrix Element      Quantity     Description
=========================================================================
        [1][1]             E     Young's modulus
        [2][1]             G     Shear modulus
        [3][1]            Fy     Yield stress
        [4][1]            ET     Tangent Young's Modulus
        [5][1]            nu     Poission's ratio
        [6][1]       density     Material density
        [7][1]            Fu     Ultimate stress
        [8][1]    thermal[0]     x-direction coeff. of thermal expansion
        [9][1]    thermal[1]     y-direction coeff. of thermal expansion
       [10][1]    thermal[2]     z-direction coeff. of thermal expansion
```

**Note :** An engineer should be careful to make sure that a particular material or section property is used in a way that is consistent with the actual material/section type. For example, `GetSection()` will return a *thickness* component for all element types, even though the concept does not apply to two- and three-dimensional beam finite elements.

| Name | Description | Units ($M^\alpha.L^\beta.T^\gamma$) |
|---|---|---|
| E | Young's modulus of elasticity | $M^1 \cdot L^{-1} \cdot T^{-2}$ |
| Et | Tangent modulus of elasticity | $M^1 \cdot L^{-1} \cdot T^{-2}$ |
| poisson | Poisson's ratio. | $M^0 \cdot L^0 \cdot T^0$ |
| density | Material density. | $M^0 \cdot L^0 \cdot T^0$ |
| yield | Yield Stress $F_y$. | $M^0 \cdot L^0 \cdot T^0$ |
| ultimate | Ultimate Stress $F_u$. | $M^0 \cdot L^0 \cdot T^0$ |
| n | Strain Hardening Exponent | $M^0 \cdot L^0 \cdot T^0$ |
| alpha | Parameter for Ramberg-Osgood relationship | $M^0 \cdot L^0 \cdot T^0$ |
| beta | Parameter for strain hardening – beta = 0 for kinematic hardening. beta = 1 for isotropic hardening. | $M^0 \cdot L^0 \cdot T^0$ |
| ialph | Rotation constant – ialph = 0 implies none. ialph = 1 implies Hughes rotation constant. | $M^0 \cdot L^0 \cdot T^0$ |
| pen | Penalty constant. | $M^0 \cdot L^0 \cdot T^0$ |

Table 5.2: **Material Properties**

## 5.6 Boundary Conditions

The boundary condition is controled through *FixNode()* function. For each degree of freedom (DOF), an index (1 or 0) is used to indicate the fixed DOF or free DOF. The syntax for *FixNode()* is:

- `FixNode(nodeno, bc_vector);` Here `nodeno` is the node number to which the boundary condition will be applied, `bc_vector` = $[bc_1, bc_2, bc_3, \cdots]$ is either a $(1 \times 3)$ matrix or a $(1 \times 6)$ matrix. $bc_i = 1$ or 0, i is the ith degree of freedom (dof). $bc_i = 1$, stands for constraint for that dof, while $bc_i = 0$ for no constraint for that dof.

**Short Example :** This script fixes the boundary conditions of a finite element mesh at nodes 1 through 5.

ABBREVIATED INPUT FILE

```
/* [a] : Apply boundary conditions */

   dx = 1; dy = 1 ; dz = 1;
   rx = 0; ry = 0 ; dz = 0;
```

```
bcond = [ dx, dy, dz, rx, ry, rz ];

for( iNode = 1; iNode <= 5; iNode = iNode + 1 ) {
     FixNode ( iNode , bcond );
}
```

We have used the variables dx, dy, and dz to represent translational displacements in the x, y, and z directions, respectively, and the variables rx, ry, and rz for rotational displacement about the x, y, and z axes. A zero value of the matrix element means that the degree of freedom remain unrestrained. A non-zero value means that full-fixity applies to the degree of freedom. Hence, nodes 1 through 5 are fixed in their translational degrees of freedom, and pinned in the three rotational degrees of freedom.

## 5.7 External Nodal Loads

External nodal loads are specified with the function `NodeLoad(nodeno, load_vector);`.

- The function `NodeLoad(nodeno, load_vector);` may be used to apply external loads to node number `nodeno`. Here `load_vector` $= [F_x, F_y, M_{xy}]$ for two-dimensional problems, and `load_vector` $= [F_x, F_y, F_z, M_{xy}, M_{yz}, M_{zx}]$ for three-dimensional problems.

**Short Example :** The following script adds two translational forces, and one moment to nodes 1 through 5 in a two-dimensional finite element mesh.

ABBREVIATED INPUT FILE

```
FxMax = 1000.0 lbf; Fy = -1000.0 lbf; Mz = 0.0 lb*in;

for( iNode = 1; iNode <= 5; iNode = iNode + 1 ) {
     Fx = (iNode/5)*FxMax;
     NodeLoad( iNode , [ Fx, Fy, Mz ]);
}
```

External nodal loads are applied in the x-direction, beginning at `200 lbf` at node 1, and increasing linearly to `1000 lbf` at node 5. A gravity load, `Fy = -1000.0 lbf` is applied to each of the nodes 1 through 5.

## 5.8 Stiffness, Mass and External Loading Matrices

The next step, after the details of the finite element mesh have been fully specified, is to calculate the finite element properties, and to assemble them into an equilibrium system. For structural finite element analyses, this step involves calculation of the stiffness and mass matrices, and an external load vector. Three finite element library functions, `Stiff()`, `Mass()`, and `ExternalLoad()` for these purposes:

- `Mass([massflag]);` where massflag is either `[1]` or `[-1]`. Therefore the Mass() command can be used as mass = Mass([1]); mass = Mass([-1]); or more explicitly, as, Lumped = [1]; mass = Mass(Lumped); Consist = [-1]; mass = Mass(Consist);
- `Stiff()` Stiff(); calculate the linear elastic stiffness matrix.
- `ExternalLoad()` ExternalLoad(); calculate the external nodal loads.

**Short Example :** In the following script of code, `mass` is the global mass matrix, `stiff` is the global stiffness matrix, and `eload` is a vector of external nodal loads applied to the finite element global degrees of freedom.

ABBREVIATED INPUT FILE

```
/* [a] : Form Mass matrix, stiffness matrix, and external load vector */

   mass  = Mass();
   stiff = Stiff();
   eload = ExternalLoad();
```

**Note :** The three functions `Mass()`, `Stiff()`, and `ExternalLoad()` should be called only after the function call to `EndMesh()` – for details on the expected input-file layout, see Section 5.2.

## 5.9 Internal Loads

In the development of many numerical/finite element algorithms, we need to know the distribution of externally applied nodal forces that will produce a given displacement pattern.

- `InternalLoad(displacement)` : Calculate the internal nodal loads for given displacement. Where displacement is a matrix vector contains nodal displacement.
- `InternalLoad(displacement, IncrementalDisplacement)` The two argument version of `InternalLoad()` computes the internal nodal load increments for a given displacement and displacement increment. Here `displacement` and `IncrementalDisplacement` are the matrix vectors for the nodal displacement and incremental displacement. This is function is designed for nonlinear analysis.

**Short Example :** In Part `[a]` of the following script we: (a) call `Copy()` to make a copy of the global stiffness matrix; (b) decompose the stiffness matrix into a product of upper and lower triangular matrices; (c) compute the external load vector; (d) solve the system of equations via forward and backward substitution, and finally, (e) setup a delta displacement vector with the first entry equal to 0.0001, and all other entries equal to zero.

ABBREVIATED INPUT FILE

```
/* [a] : Compute initial displacement and displacement increment */

   lu    = Decompose(Copy(stiff));
   eload = ExternalLoad();
   displ = Substitution(lu, eload0);

   delta_displ = Zero([size, 1]);
   delta_displ [1][1] = 0.0001;

/* [b] : Internal load for initial displacement plus displacement increment */

   iload = InternalLoad(displ, delta_displ);
```

In the second half of this script, `InternalLoad()` computes the internal nodal loads for the displacement vector `displ` plus `delta_displ`.

## 5.10 Retrieving Information from ALADDIN

We have written two families of functions that query ALADDIN's database for information on the finite element mesh, and behavior of a particular finite element model. Together, these functions may be used for design rule checking, and for the preparation of matrix output that is compatible with input to MATLAB graphics. The latter could be a plot of the finite element mesh, contours of displacements, stresses, and so forth.

The functions that will retrieve information on the finite element mesh are:

- `GetCoord([nodeno])` takes as its input a $(1 \times 1)$ matrix containing the node number. For two-dimensional finite element problems, `GetCoord()` returns a $(1x2)$ matrix ( m[1][1], m[1][2]) = (x,y). A $(1 \times 3)$ matrix is returned for three-dimensional elements (m[1][1], m[1][2], m[1][3]) = (x,y,z).

- `GetDof([nodeno])` is short for "get degree of freedom." Get the matching global degrees of freedom number for a node. It returns a (1xn) matrix, n equals to the number of degrees of freedom in the node. For two-dimensional beam-column elements, n = 3, and (m[1][1], m[1][2], m[1][3]) = ( dof_dx, dof_dy, dof_rz); FOr the three-dimensional frame element, n = 6, and (m[1][1],m[1][2],m[1][3],m[1][4],m[1][5],m[1][6])

= (dof_dx,dof_dy,dof_dz,dof_rx,dof_ry,dof_rz). A negative number in the output matrix indicates the associated degree of freedom is fixed.

- `GetNode([elmtno])` retrieves a matrix $(1 \times n)$ of node numbers connected to finite element `elmtno`. `n` equals to the number of nodes in the element. For example, n = 2 for 2D and 3D beam element, and n = 4 for the 4-node shell element. The matrix will be in the same order as you input the nodes connection in AddElmt()

Similarly, the functions for retrieving information on the behavior of a finite element model are:

- `GetDispl([nodeno, displ_m])` is short for "get displacement," and takes as input, a node number and the calculated displacement matrix. Get the structural nodal displacements. It returns a (1xn) matrix, nequals to the number of degrees of freedom in the node. For two-dimensional beam elements, n = 3, and (m[1][1], m[1][2], m[1][3]) = (displ_x,displ_y,rot_z). Similarly, for three-dimensional frame elements, n = 6, and ( m[1][1], m[1][2], m[1][3], m[1][4], m[1][5], m[1][6]) = (displ_x, displ_y, displ_z, rot_x, rot_y, rot_z).

- `GetStress([elmtno],displacement_matrix)` accepts as its input, ana element number and the calculated displacement matrix. This function gets the nodal forces for the element.

  In ALADDIN Version 1.0, this function only works for 2D and 3D beam elements. It returns a (mxn) matrix, m equals to the number of nodes in the element, n equals to the number of degrees of freedom in a node. For example, (mxn)=(2x3) for 2D beam element,

  ```
               1       2       3
   node 1    Fx1     Fy1     Mz1
   node 2    Fx2     Fy2     Mz2

  (mxn)=(2x6) for 2D beam element,

               1       2       3       4       5       6
   node 1    Fx1     Fy1     Fz1     Mx1     My1     Mz1
   node 2    Fx2     Fy2     Fz2     Mx2     My2     Mz2
  ```

  The order of the nodes 1,2... is defined in the nodes connection matrix in AddElmt().

These functions will be demonstrated in Chapter 6.

## 5.11 Library of Finite Elements

Currently, the following finite elements are available:

**PLANE_STRESS/PLANE_STRESS :** Two dimensional 4-node plane stress/plane strain element.

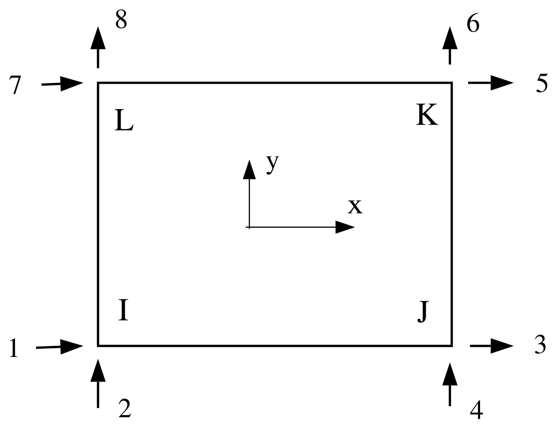


Figure 5.2: Schematic of Plane Stress - Plane Strain Finite Element

May be used to model linear elastic materials.

**Section Properties and Material Properties :** Young's modulus `E`; Poisson's ratio $\nu$; Density $\rho$ for dynamic analyses; Element type = `PLANE_STRESS` (or `PLANE_STRAIN`). No section properties are needed.

**Example :**

ABBREVIATED INPUT FILE

```
beam_length  = 8 m;
beam_width   = 1 m;
beam_height  = 2 m;

nodeno = 1; elmtno = 0;

while(elmtno <= 3) {
   elmtno = elmtno + 1;
   AddElmt( elmtno, [nodeno+3, nodeno+1, nodeno+2, nodeno+4], "name_of_elmt_attr");
   nodeno = i + 2;
}

....... input statements removed ....
```

**FRAME_2D** : Two dimensional 2-node frame (or beam/column) element.

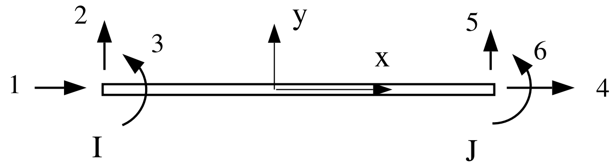


Figure 5.3: Schematic of Two Dimensional Frame Element

Each node has two translational, and one rotational, degree of freedom. May be used for modeling of linear elastic materials.

**Section Properties and Material Properties :** The section properties are; Moment of inertia `Izz` and cross-section `area` – alternatively, the cross section are is computed from the section `width` (or `bf`) times the `depth`. The material properties are Young's modulus `E`, Poisson's ratio $\nu$, and density $\rho$. Note – the density may be omitted when a default I-beam section is referenced from the AISC sections header file `section.h`. Element type = `FRAME_2D`.

**Example :**

ABBREVIATED INPUT FILE

```
end1 = 4*floorno + bayno;
end2 = end1 + 1;

AddElmt( elmtno, [ end1 , end2 ], "mybeam");

....... input code removed ........

ElementAttr("floorbeam") { type     = "FRAME_2D";
                           section  = "mysection2";
                           material = "mymaterial";
                         }

SectionAttr("floorsection2") { Izz      = 1600.3 in^4;
                               Iyy      = 66.2   in^4;
                               depth    = 21.0     in;
                               width    = 8.25     in;
                               area     = 21.46  in^2;
                             }

MaterialAttr("floormaterial") { density = 0.1024E-5 lb/in^3;
                                poisson = 0.25;
                                yield   = 36.0   ksi;
                                E       = 29000  ksi;
                              }
```

**FRAME_3D** : Three dimensional 2-node frame (or beam/column) element.

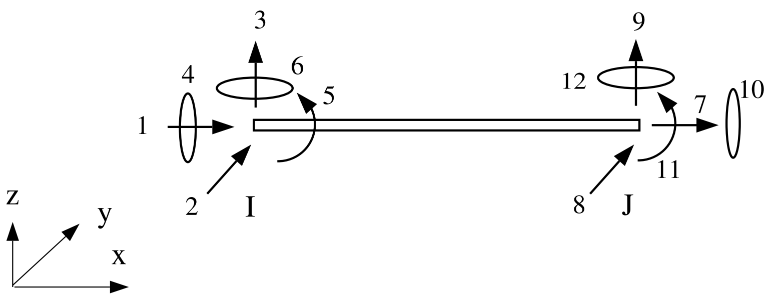


Figure 5.4: Schematic of Two Dimensional Frame Element

Each node has three translational, and three rotational, degrees of freedom. May be used for modeling of linear elastic materials.

**Section Properties and Material Properties :** The section properties include moments of inertia `Izz` and `Iyy`, and the cross-section `area`. The cross-section area may also be computed from the section `depth` times its `width` (or `bf`). The torsional constant and radius of gyration are defined by parameters `J` and `rT`, respectively. The material properites are Young's modulus `E`, Poisson's ratio $\nu$, and the density $\rho$. Note – the density may be omitted when a default I-beam section is referenced from the AISC sections header file `section.h`. The element type is `FRAME_3D`.

**Example :**

ABBREVIATED INPUT FILE

```
elmtno = 0;
while(elmtno < 4) {
      elmtno = elmtno + 1;
      AddElmt( elmtno, [elmtno, elmtno + 1], "floorelmt");
}

....... code deleted ....

ElementAttr("floorelmt") { type     = "FRAME_3D";
                           section  = "mysection";
                           material = "mymaterial";
                         }

SectionAttr("floorsection") { Izz     = 13824 in^4;
                              Iyy     =  3456 in^4;
                              area    =   288 in^2;
                              depth   =    24 in;
                              width   =    12 in;
                            }

MaterialAttr("floormaterial") { .... same as for FRAME_2D example .... }
```

**GENERAL SHELL**: Three dimensional 4 and 8-node five degree of freedom general shell elements, linear/nonlinear elastic-plastic materials.

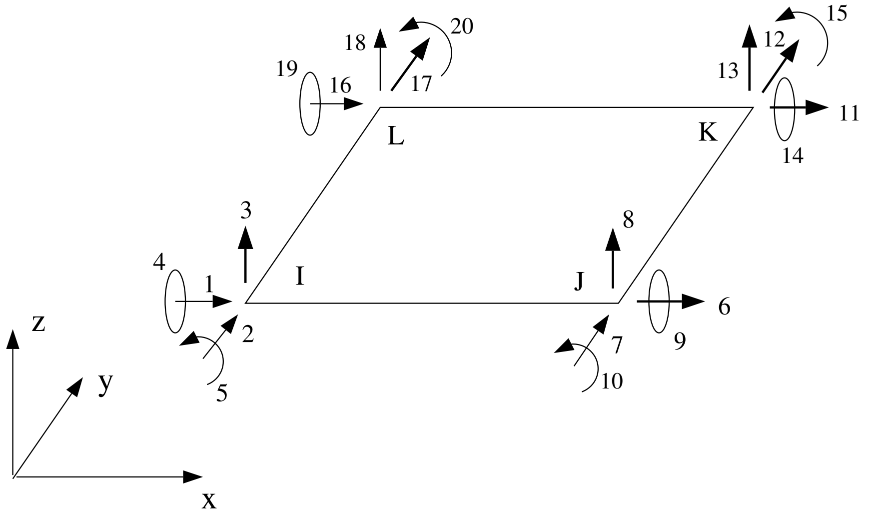


Figure 5.5: Schematic of General Shell Element

The element formulation is described in Chen and Austin [10].

**Section Properties and Material Properties :** The section attribute is `thickness` alone. The material properties are Young's modulus `E`, Poisson's ratio $\nu$ and density $\rho$ (for dynamic analysis). The following parameters apply for nonlinear analysis – Yield stress `yield`, stress-strain parameters `n`, `alpha` and `beta`, and `type` (Ramberg-Osgood or Bi-Linear or ELASTIC_PLASTIC material).

The material types are `ELASTIC`, `ELASTIC_PLASTIC` and `ELASTIC_PERFECTLY_PLASTIC`.

The element types for the four node and eight node shell elements are `SHELL_4N` and `SHELL_8N`.

**Example :**

ABBREVIATED INPUT FILE

```
elmtno = 1;
node_connec = [ 1, 2, 3, 4, 12, 13, 14, 15];
AddElmt(elmtno, node_connec, "name_of_elmt_attr");

..... input code removed .....

ElementAttr("name_of_elmt_attr") { type     = "SHELL_8N";
                                   section  = "mysection";
                                   material = "ELASTIC";
                                 }
MaterialAttr("ELASTIC") { density = 1.0 lb/in^3;
                          poisson = 0.25;
```

```
                    yield   = 36000;
                    E       = 3E+7 psi;
                  }

SectionAttr("mysection") { thickness =   1 in; }
```

**SHELL WITH DRILLING DEGREE OF FREEDOM :** A four node flat shell element with six degrees of freedom per node. It may be used for modeling of linear elastic materials.

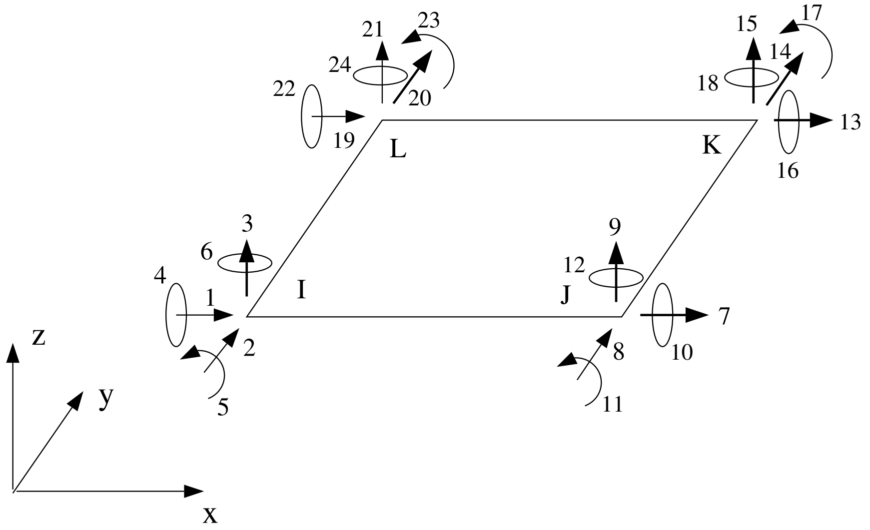


Figure 5.6: Schematic of Flat Shell Finite Element

The theoretical formulation of this shell finite element, and numerical examples of its performance may be found in the masters thesis of Lanheng Jin [15].

**Section Properties and Material Properties :** The section attribute is `thickness` alone. The material properties are Young's modulus `E`, Poisson's ratio $\nu$, and Density $\rho$ (for dynamic analyses). The element has two modeling parameters `ialpha` and `pen`. The element type is `SHELL_4NQ`.

**Example :**

ABBREVIATED INPUT FILE

```
AddElmt( elmtno+1, [ a+1, b+1, b+2, a+2 ], "bridgegirder" );

..... input code removed .....

ElementAttr("bridgegirder") { type     = "SHELL_4NQ";
                              section  = "girder_flange";
```

```
                                 material = "STEEL3";
                               }

SectionAttr( "girder_flange" ) { thickness = 1.100 in; }
```

STEEL3 is a material property predefined in material.h, and loaded into the ALADDIN database during the program's startup procedure.

**PLATE**: Four node discrete kirchoff quadrilateral (DKQ) plate element. Each node has three degrees of freedom – two rotations, and a lateral displacement, as shown in Figure 5.7.

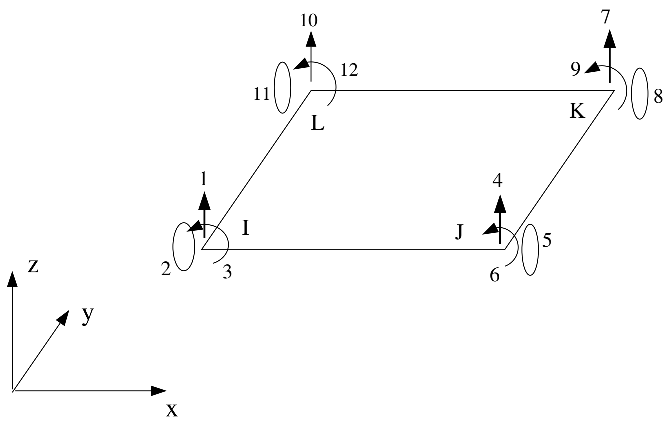


Figure 5.7: Schematic of Plate Finite Element

**Section Properties and Material Properties :** The section property is plate thickness. The material properties include Young's modulus E, and Poisson's ratio $\nu$. The element type is DKT_PLATE.

**Example :**

ABBREVIATED INPUT FILE

```
connect = [ 1, 2, 3, 4];
AddElmt( elmtno, connect, "name_of_elmt_attr");

...... input statements removed .....

ElementAttr("name_of_elmt_attr") { type     = "DKT_PLATE";
                                   section  = "mysection";
                                   material = "ELASTIC";
                                   map ldof [1,2,3] to gdof [2, 4, 6];
                                 }

SectionAttr("mysection") { thickness = 2.0 in; }
```

```
MaterialAttr("ELASTIC")  { density = 150 lb/ft^3;
                           poisson = 0.3;
                           yield   = 36000;
                           E       = 29000 psi;
```

Currently, this element does not have a mass matrix.

# Chapter 6

# Input Files for Finite Element Analysis Problems

In this chapter we exercise ALADDIN's finite element capabilities by working through four problem in detail. The finite element problems are:

[1] A linear static analysis of a five story moment resistant frame. The frame supports gravity loads plus a moderate lateral load.

[2] A linear time-history earthquake analysis of the five story frame described in Item [1]. The time-history response is generated by a 1979 El Centro ground motion.

[3] An AASHTO working stress design of a composite highway bridge girder, with rule checking built into the input file.

[4] A linear elastic three dimensional analysis of 2-span highway bridge. Again, the bridge is modeled with the four node shell element. Moment envelopes are computed for a gravity loads plus a moving truck live load.

The analyses in problems [3] and [4] include moving live loads. The application of ALADDIN to the solution of nonlinear static analyses with an 8-node shell element is reported in Chen and Austin [10].

## 6.1 Linear Static Analyses

### 6.1.1 Analysis of Five Story Moment Resistant Frame

**Description of Problems :** We begin with a linear static analysis of a five story steel moment resistant frame. A front elevation view of the frame with dimensions and preliminary section sizes is shown in Figure 6.1. We will assume that this frame is one in a row of frames spaced at `20 ft` centers, as shown in Figure 6.2.

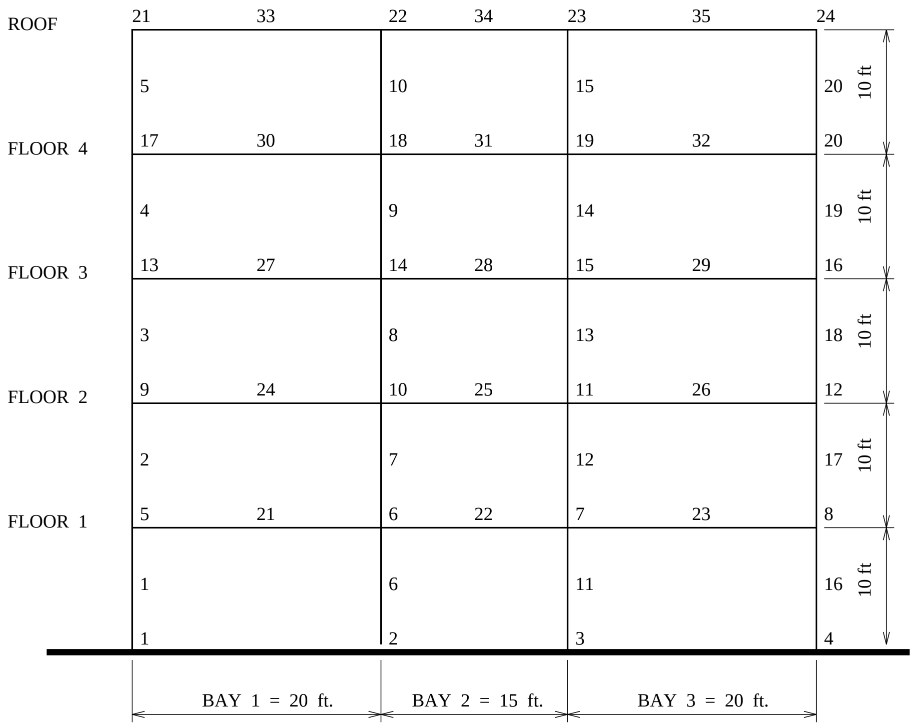


Figure 6.1: Elevation View of Five Story Planar Steel Frame

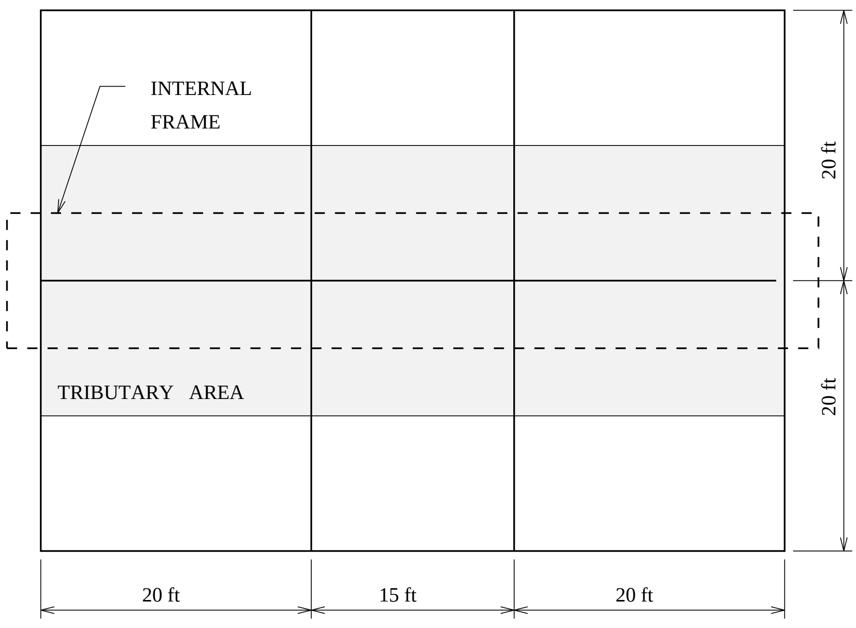


Figure 6.2: Plan View of Five Story Building

The frame is constructed with two steel section types, and one material type. The columns are of type UC $12 \times 12$, with $I_{zz} = 1541.9\ in^4$, $I_{yy} = 486.3\ in^4$, and cross section area 47.38 $in^2$. The beams are of section type UB $21 \times 81/4$, with $I_{zz} = 1600.3\ in^4$, $I_{yy} = 66.2\ in^4$, and cross section area 21.46 $in^2$. The beams and columns have Young's Modulus 29000 ksi, and yield stress 36.0 ksi.

The frame will be analyzed for gravity loads plus a moderate lateral earthquake load. Wind and other loads are omitted for simplicity. The specified dead load is `80 lbf/ft`$^2$, and the specified live load `40 lbf/ft`$^2$ for the four floors, and `20 lbf/ft`$^2$ for the roof. We will assume that the tributary area for gravity loads is rectangular in shape, with a uniform load distribution along each girder span of the frame. With the frames spaced `20 ft` apart, it follows that floor gravity loads are `0.200` kips/in, and `0.1667` kips/in on the roof. Total gravity loads are `663800 lbf`.

In this first release of ALADDIN, external loads may only be applied to the nodes. Consequently, formulae for the fixed-end moments due to gravity loads must be explicitly included in the data file. If the loading per unit length is $w$, and $L$ is the span length of the frame bay, then the fixed end shear force is $wL/2$, and the fixed end moment is $wL^2/12$.

Moderate earthquake ground motions are modeled with a psuedo-static lateral force equal to 10% of total gravity loads (i.e. 63800 lbf). The lateral forces are distributed over the height of the frame as shown in Figure 6.3.

We will assume full fixity for the bases of the columns. Each node of the building frame will be modeled with two translational and one rotational degree of freedom. Axial forces in the beam elements are removed by lumping the horizontal degrees of freedom at each floor level. Together these assumptions imply 5 displacement and 4 rotational degrees of freedom per floor level, leading to $(45 \times 45)$ global stiffness matrix.

**Input File :** The finite element model for the five story moment-resistant building frame, with external loads, is defined and solved in the following `[a]` to `[h]` part input file.

START OF INPUT FILE

```
/*
 *  ===================================================================
 *  Analysis of Five Story Steel Moment Resistant Frame
 *
 *  Written By: Mark Austin                                October, 1994
 *  ===================================================================
 */

/* [a] : Setup problem specific parameters */

   NDimension         = 2;
   NDofPerNode        = 3;
   MaxNodesPerElement = 2;
```

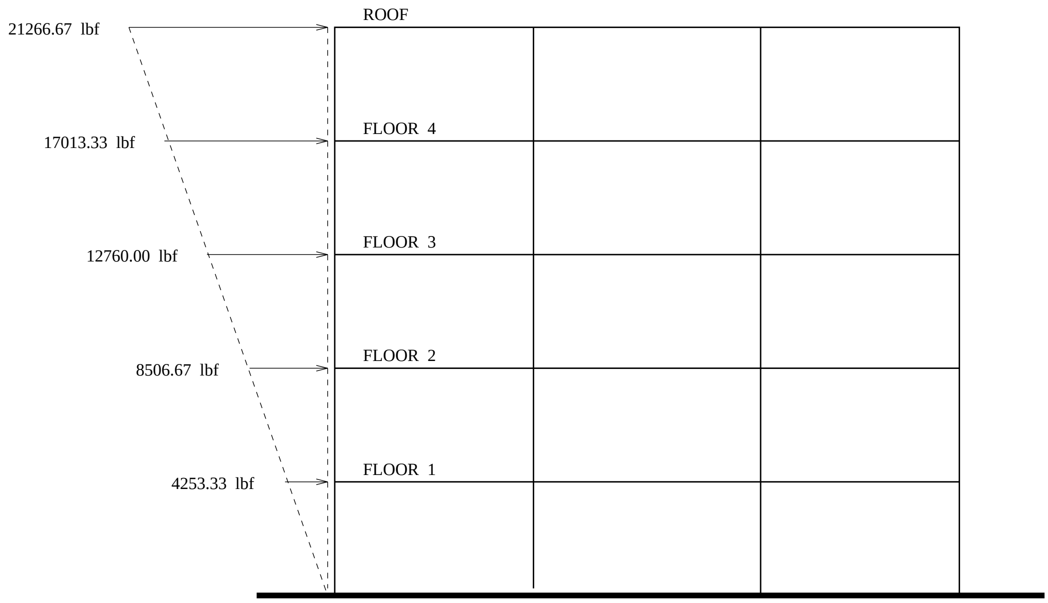


Figure 6.3: Distribution of Lateral Loads for Moderate Earthquake Ground Motions

```
   StartMesh();

/* [b] : Generate two-dimensional grid of nodes */

   node = 0;
   for( y = 0 ft; y <= 50 ft; y = y + 10 ft ) {
      for( x = 0 ft; x <= 55 ft; x = x + 20 ft ) {

        /* [b.1] : adjust column spacing for central bay */

           if(x == 40 ft) {
              x = x - 5 ft;
           }

        /* [b.2] : add new node to finite element mesh */

           node = node + 1;
           AddNode(node, [ x, y ] );
      }
   }

/* [c] : Attach column elements to nodes */

   elmtno = 0;
   for (  colno = 1;   colno <= 4;   colno = colno + 1) {
   for (floorno = 1; floorno <= 5; floorno = floorno + 1) {
        elmtno = elmtno + 1;

        end1 = 4*(floorno - 1) + colno;
        end2 = end1 + 4;
```

```
        AddElmt( elmtno, [ end1 , end2 ], "mycolumn");
   }
   }

/* [d] : Attach beam elements to nodes */

   for (floorno = 1; floorno <= 5; floorno = floorno + 1) {
   for (  bayno = 1;   bayno <= 3;     bayno = bayno + 1) {

        end1 = 4*floorno + bayno;
        end2 = end1 + 1;

        elmtno = elmtno + 1;
        AddElmt( elmtno, [ end1 , end2 ], "mybeam");
   }
   }

/* [e] : Define section and material properties */

   ElementAttr("mycolumn") { type     = "FRAME_2D";
                             section  = "mysection1";
                             material = "mymaterial";
                           }

   ElementAttr("mybeam") { type     = "FRAME_2D";
                           section  = "mysection2";
                           material = "mymaterial";
                           }

   SectionAttr("mysection1") { Izz       = 1541.9 in^4;
                               Iyy       =  486.3 in^4;
                               depth     =   12.0 in;
                               width     =   12.0 in;
                               area      =   47.4 in^2;
                             }

   SectionAttr("mysection2") { Izz       = 1600.3 in^4;
                               Iyy       =   66.2 in^4;
                               depth     =   21.0 in;
                               width     =   8.25 in;
                               area      =  21.46 in^2;
                             }

   MaterialAttr("mymaterial") { density = 0.1024E-5 lb/in^3;
                                poisson = 0.25;
                                yield   = 36.0   ksi;
                                E       = 29000  ksi;
                              }

/* [f] : Apply full-fixity to columns at foundation level */

   for(nodeno = 1; nodeno <= 4; nodeno = nodeno + 1) {
       FixNode( nodeno, [ 1, 1, 1 ]);
```

```
   }

   LinkNode([  5,  6,  7,  8 ], [ 1, 0, 0] );
   LinkNode([  9, 10, 11, 12 ], [ 1, 0, 0] );
   LinkNode([ 13, 14, 15, 16 ], [ 1, 0, 0] );
   LinkNode([ 17, 18, 19, 20 ], [ 1, 0, 0] );
   LinkNode([ 21, 22, 23, 24 ], [ 1, 0, 0] );

/* [g] : Compute equivalent nodal loads for distributed "dead + live" loads plus  */
/*       lateral wind loads                                                        */

   dead_load       = 80 lbf/ft^2;
   floor_live_load = 40 lbf/ft^2;
   roof_live_load  = 20 lbf/ft^2;
   frame_spacing   = 20 ft;

   for (floorno = 1; floorno <= 5; floorno = floorno + 1) {

   /* [g.1] : compute floor-level (and roof-level) uniform loads */

      live_load = floor_live_load;
      if( floorno == 5) {
          live_load = roof_live_load;
      }
      uniform_load = (dead_load + live_load)*(frame_spacing);

      for (colno = 1;   colno <= 4;     colno = colno + 1) {

         Fx = 0.0 lbf; Fy = 0.0 lbf; Mz = 0.0  lb*in;

      /* [g.2] : compute fixed end shear force for dead/live loads */

         if( colno == 1 || colno == 4) {
             Fy = -(uniform_load)*(20 ft)/2;
         }

         if( colno == 2 || colno == 3) {
             Fy = -(uniform_load)*(35 ft)/2;
         }

      /* [g.3] : compute fixed end moments for dead/live loads */

         if( colno == 1 ) {
             Mz = -(uniform_load)*(20 ft)*(20 ft)/12;
         }

         if( colno == 2 ) {
             Mz =  (uniform_load)*((20 ft)^2 - (15 ft)^2)/12;
         }

         if( colno == 3 ) {
             Mz = -(uniform_load)*((20 ft)^2 - (15 ft)^2)/12;
         }
```

```
        if( colno == 4 ) {
            Mz =  (uniform_load)*(20 ft)*(20 ft)/12;
        }

     /* [g.4] : compute horizontal force due to lateral loads */

        if(colno == 1 ) {
           Fx = 63800*(floorno/15)*(1 lbf);
        }

        nodeno = 4*floorno + colno;
        NodeLoad( nodeno, [ Fx, Fy, Mz ]);
     }
  }

/* [h] : Compile and Print Finite Element Mesh */

  EndMesh();
  PrintMesh();

/* [i] : Compute "stiffness" and "external load" matrices */

  eload = ExternalLoad();
  stiff = Stiff();

  displ = Solve(stiff, eload);

  SetUnitsType("US");
  PrintDispl(displ);
  PrintStress(displ);
  quit;
```

Points to note are:

[1] In part `[a]` we specify that this will be a two-dimensional analysis. The maximum number of degrees of freedom per node will be three, and the maximum number of nodes per element will be two. The parameters `NDimension`, `NDofPerNode`, and `MaxNodesPerElement` are used by ALADDIN to assess memory requirements for the problem storage and solution.

[2] We use a nested `for()` loop and a single `if()` statement to generate the planar layout of 24 finite element nodes. Before the boundary conditions are applied, the structure has 72 degrees of freedom. In Section `[f]` we apply full-fixity to each column at the foundation level – this reduces degrees of freedom from 72 to 60.

**Abbreviated Output File :** The output file contains summaries of the mass and stiffness matrices for the shear building, and abbreviated details of the external loading, `"myload"`, and the response matrix `"response"`.

START OF ABBREVIATED OUTPUT FILE

```
==========================================
Title : DESCRIPTION OF FINITE ELEMENT MESH
==========================================

 Problem_Type:  Static Analysis

=======================
Profile of Problem Size
=======================

   Dimension of Problem        =      2

   Number Nodes                =     24
   Degrees of Freedom per node =      3
   Max No Nodes Per Element    =      2

   Number Elements             =     35
   Number Element Attributes   =      2
   Number Loaded Nodes         =     20
   Number Loaded Elements      =      0

------------------------------------------------------------
Node#      X_coord           Y_coord         Tx    Ty    Rz
------------------------------------------------------------

    1    0.00000e+00 ft    0.00000E+00 ft    -1    -2    -3
    2    2.00000e+01 ft    0.00000E+00 ft    -4    -5    -6
    3    3.50000e+01 ft    0.00000E+00 ft    -7    -8    -9
    4    5.50000e+01 ft    0.00000E+00 ft   -10   -11   -12
    5    0.00000e+00 ft    1.00000E+01 ft     1     6     7

   ...... details of nodal coordinates and modeling dof removed ....

   19    3.50000e+01 ft    4.00000E+01 ft     4    34    35
   20    5.50000e+01 ft    4.00000E+01 ft     4    36    37
   21    0.00000e+00 ft    5.00000E+01 ft     5    38    39
   22    2.00000e+01 ft    5.00000E+01 ft     5    40    41
   23    3.50000e+01 ft    5.00000E+01 ft     5    42    43
   24    5.50000e+01 ft    5.00000E+01 ft     5    44    45

--------------------------------------------------------------------
Element#       Type       node[1]       node[2]    Element_Attr_Name
--------------------------------------------------------------------

       1     FRAME_2D          1           5            mycolumn
       2     FRAME_2D          5           9            mycolumn
       3     FRAME_2D          9          13            mycolumn
       4     FRAME_2D         13          17            mycolumn

      ...... details of element connectivity removed ....

      33     FRAME_2D         21          22              mybeam
      34     FRAME_2D         22          23              mybeam
```

```
      35     FRAME_2D         23         24              mybeam

---------------------
Element Attribute Data :
---------------------

ELEMENT_ATTR No.   1  : name = "mycolumn"
                      : section = "mysection1"
                      : material = "mymaterial"
                      : type = FRAME_2D
                      : gdof [0] =    1 : gdof[1] =    2 : gdof[2] =    3
                      : Young's Modulus =  E =        2.900e+04 ksi
                      : Yielding Stress = fy =        3.600e+01 ksi
                      : Poisson's ratio = nu =        2.500e-01
                      : Density         =        1.024e-06 lb/in^3
                      : Inertia Izz     =        1.542e+03 in^4
                      : Area            =        4.740e+01 in^2

ELEMENT_ATTR No.   2  : name = "mybeam"
                      : section = "mysection2"
                      : material = "mymaterial"
                      : type = FRAME_2D
                      : gdof [0] =    1 : gdof[1] =    2 : gdof[2] =    3
                      : Young's Modulus =  E =        2.900e+04 ksi
                      : Yielding Stress = fy =        3.600e+01 ksi
                      : Poisson's ratio = nu =        2.500e-01
                      : Density         =        1.024e-06 lb/in^3
                      : Inertia Izz     =        1.600e+03 in^4
                      : Area            =        2.146e+01 in^2

EXTERNAL NODAL LOADINGS
------------------------------------------------
Node#      Fx (lbf)      Fy (lbf)    Mz (lbf.in)
------------------------------------------------
    5      4253.33     -24000.00    -960000.00
    6         0.00     -42000.00     420000.00
    7         0.00     -42000.00    -420000.00
    8         0.00     -24000.00     960000.00
    9      8506.67     -24000.00    -960000.00
   10         0.00     -42000.00     420000.00
   11         0.00     -42000.00    -420000.00
   12         0.00     -24000.00     960000.00
   13     12760.00     -24000.00    -960000.00
   14         0.00     -42000.00     420000.00
   15         0.00     -42000.00    -420000.00
   16         0.00     -24000.00     960000.00
   17     17013.33     -24000.00    -960000.00
   18         0.00     -42000.00     420000.00
   19         0.00     -42000.00    -420000.00
   20         0.00     -24000.00     960000.00
   21     21266.67     -20000.00    -800000.00
   22         0.00     -35000.00     350000.00
   23         0.00     -35000.00    -350000.00
   24         0.00     -20000.00     800000.00
```

```
============= End of Finite Element Mesh Description ==============

-------------------------------------------------------------
 Node                          Displacement
  No            displ-x           displ-y            rot-z
-------------------------------------------------------------
 units              in                in               rad
   1         0.00000e+00       0.00000e+00       0.00000e+00
   2         0.00000e+00       0.00000e+00       0.00000e+00
   3         0.00000e+00       0.00000e+00       0.00000e+00
   4         0.00000e+00       0.00000e+00       0.00000e+00
   5         1.02973e-01      -7.40873e-03      -1.23470e-03
   6         1.02973e-01      -1.64731e-02      -6.24404e-04
   7         1.02973e-01      -1.90281e-02      -8.22556e-04
   8         1.02973e-01      -1.27863e-02      -7.58887e-04
   9         2.55749e-01      -1.34750e-02      -1.19639e-03

   ....... details of displacements removed .....

  21         5.62354e-01      -2.29345e-02      -6.66741e-04
  22         5.62354e-01      -4.88997e-02      -8.48912e-05
  23         5.62354e-01      -5.50729e-02      -3.30685e-04
  24         5.62354e-01      -3.63403e-02       6.32278e-05

MEMBER FORCES
-------------------------------------------------------------------------------

Elmt No   1 :
Coords (X,Y) = (     0.000 in,     60.000 in)
exx =  -6.17394e-05 , curva =   -0.00040508 , gamma =  -3.10985e-04

 Fx1 =   8.48669e+04 lbf  Fy1 =   8.97143e+03 lbf  Mz1 =   9.98366e+05 lbf.in
 Fx2 =  -8.48669e+04 lbf  Fy2 =  -8.97143e+03 lbf  Mz2 =   7.82053e+04 lbf.in

 Axial Force : x-direction =  -8.48669e+04 lbf
 Shear Force : y-direction =   8.97143e+03 lbf

Elmt No   6 :
Coords (X,Y) = (   240.000 in,     60.000 in)
exx =  -1.37276e-04 , curva =   -0.00020486 , gamma =  -7.05134e-04

 Fx1 =   1.88700e+05 lbf  Fy1 =   2.03420e+04 lbf  Mz1 =   1.45319e+06 lbf.in
 Fx2 =  -1.88700e+05 lbf  Fy2 =  -2.03420e+04 lbf  Mz2 =   9.87850e+05 lbf.in

 Axial Force : x-direction =  -1.88700e+05 lbf
 Shear Force : y-direction =   2.03420e+04 lbf

Elmt No  11 :
Coords (X,Y) = (   420.000 in,     60.000 in)
exx =  -1.58567e-04 , curva =   -0.00026987 , gamma =  -5.77161e-04

 Fx1 =   2.17967e+05 lbf  Fy1 =   1.66502e+04 lbf  Mz1 =   1.30552e+06 lbf.in
 Fx2 =  -2.17967e+05 lbf  Fy2 =  -1.66502e+04 lbf  Mz2 =   6.92505e+05 lbf.in
```

```
 Axial Force : x-direction =  -2.17967e+05 lbf
 Shear Force : y-direction =   1.66502e+04 lbf

Elmt No  16 :
Coords (X,Y) = (   660.000 in,     60.000 in)
exx =  -1.06552e-04 , curva =   -0.00024898 , gamma =  -6.18281e-04

 Fx1 =   1.46467e+05 lbf  Fy1 =   1.78364e+04 lbf  Mz1 =   1.35297e+06 lbf.in
 Fx2 =  -1.46467e+05 lbf  Fy2 =  -1.78364e+04 lbf  Mz2 =   7.87403e+05 lbf.in

 Axial Force : x-direction =  -1.46467e+05 lbf
 Shear Force : y-direction =   1.78364e+04 lbf
```

---

**Remark 1 :** We have printed the internal element forces for the columns between the foundation and the first floor level. The accuracy of the analysis can be checked by making sure the sum of column actions is balanced by the sum of external loads. In the horizontal direction we have:

```
====================================================================
External Forces                           Shear Forces in Columns
====================================================================
Roof          21,266.27 lbf
Floor 4       17,013.33 lbf               Element  1    8,971.43 lbf
Floor 3       12,760.00 lbf               Element  6   20,342.00 lbf
Floor 2        8,506.67 lbf               Element 11   16,650.20 lbf
Floor 1        4,253.33 lbf               Element 16   17,836.40 lbf
====================================================================
Total         63,800.00 lbf               Total        63,800.03 lbf
====================================================================
```

And in the vertical direction we have:

```
====================================================================
Gravity Loads                             Axial Forces in Columns
====================================================================
Roof          -110,000 lbf
Floor 4       -132,000 lbf                Element  1   -84,866.9 lbf
Floor 3       -132,000 lbf                Element  6  -188,700.0 lbf
Floor 2       -132,000 lbf                Element 11  -217,967.0 lbf
Floor 1       -132,000 lbf                Element 16  -146,467.0 lbf
====================================================================
Total         -638,000 lbf                Total       -638,000.9 lbf
====================================================================
```

**Remark 2 :** The five story moment resistant frame may be modeled as a shear structure by fixing the nodal rotations and vertical displacements at floors 1 to 5, and lumping the horizontal displacements at each floor level. These tasks are accomplished by modifying Section [f] of the input file to

```
/* [f] : Apply full-fixity to columns at foundation level */

   for(nodeno = 1; nodeno <= 4; nodeno = nodeno + 1) {
       FixNode( nodeno, [ 1, 1, 1 ]);
   }

   for(nodeno = 5; nodeno <= 24; nodeno = nodeno + 1) {
       FixNode( nodeno, [ 0, 1, 1 ]);
   }

   LinkNode([  5,  6,  7,  8 ], [ 1, 0, 0] );
   LinkNode([  9, 10, 11, 12 ], [ 1, 0, 0] );
   LinkNode([ 13, 14, 15, 16 ], [ 1, 0, 0] );
   LinkNode([ 17, 18, 19, 20 ], [ 1, 0, 0] );
   LinkNode([ 21, 22, 23, 24 ], [ 1, 0, 0] );
```

Overall displacements in the frame will now be represented with five global degrees of freedom. Details of the stiffness and load vector are:

```
MATRIX : "eload"

row/col                  1
        units
   1      lbf   4.25333e+03
   2      lbf   8.50667e+03
   3      lbf   1.27600e+04
   4      lbf   1.70133e+04
   5      lbf   2.12667e+04

SKYLINE MATRIX : "stiff"

row/col                  1             2             3             4             5
        units          N/m           N/m           N/m           N/m           N/m
   1            4.35045e+08  -2.17523e+08   0.00000e+00   0.00000e+00   0.00000e+00
   2           -2.17523e+08   4.35045e+08  -2.17523e+08   0.00000e+00   0.00000e+00
   3            0.00000e+00  -2.17523e+08   4.35045e+08  -2.17523e+08   0.00000e+00
   4            0.00000e+00   0.00000e+00  -2.17523e+08   4.35045e+08  -2.17523e+08
   5            0.00000e+00   0.00000e+00   0.00000e+00  -2.17523e+08   2.17523e+08
```

A summary of output is:

```
------------------------------------------------------------
 Node                          Displacement
  No            displ-x            displ-y             rot-z
------------------------------------------------------------
 units               in                 in               rad
   1        0.00000e+00        0.00000e+00       0.00000e+00
   2        0.00000e+00        0.00000e+00       0.00000e+00
   3        0.00000e+00        0.00000e+00       0.00000e+00
   4        0.00000e+00        0.00000e+00       0.00000e+00
   5        5.13652e-02        0.00000e+00       0.00000e+00
   6        5.13652e-02        0.00000e+00       0.00000e+00
```

```
  7         5.13652e-02      0.00000e+00      0.00000e+00
  8         5.13652e-02      0.00000e+00      0.00000e+00
  9         9.93061e-02      0.00000e+00      0.00000e+00
 10         9.93061e-02      0.00000e+00      0.00000e+00

 ...... details of displacements removed ......

 21         1.88339e-01      0.00000e+00      0.00000e+00
 22         1.88339e-01      0.00000e+00      0.00000e+00
 23         1.88339e-01      0.00000e+00      0.00000e+00
 24         1.88339e-01      0.00000e+00      0.00000e+00


MEMBER FORCES
-----------------------------------------------------------------------

Elmt No   1 :

 Axial Force : x-direction =   0.00000e+00 lbf
 Shear Force : y-direction =   1.59500e+04 lbf

Elmt No   6 :

 Axial Force : x-direction =   0.00000e+00 lbf
 Shear Force : y-direction =   1.59500e+04 lbf

Elmt No  11 :

 Axial Force : x-direction =   0.00000e+00 lbf
 Shear Force : y-direction =   1.59500e+04 lbf

Elmt No  16 :

 Axial Force : x-direction =   0.00000e+00 lbf
 Shear Force : y-direction =   1.59500e+04 lbf
```

Fixing nodal rotations increases the overall stiffness of the structure – the result is lateral displacements in the shear structure which are smaller than for the moment resistant frame (i.e. 1.8834e-01 inches versus 5.6235e-01 inches). Again, notice that shear forces across the base of the structure are balanced by the external loads (i.e 4 times 1.59500e+04 lbf = 6.3800e+04 lbf).

### 6.1.2 Working Stress Design (WSD) of Simplified Bridge

**Description of WSD Rule Checking Problem :** This example illustrates the application of ALADDIN to AASHTO Working Stress Design (WSD) code checking. We will conduct a finite element analysis of a one span simply supported composite bridge with cover plate under the girders, and then apply the rule checking. The analysis will be simplified by considering only an internal girder of the bridge system – a plan and cross sectional view of a typical bridge system is shown in Figures 6.5 and 6.5.

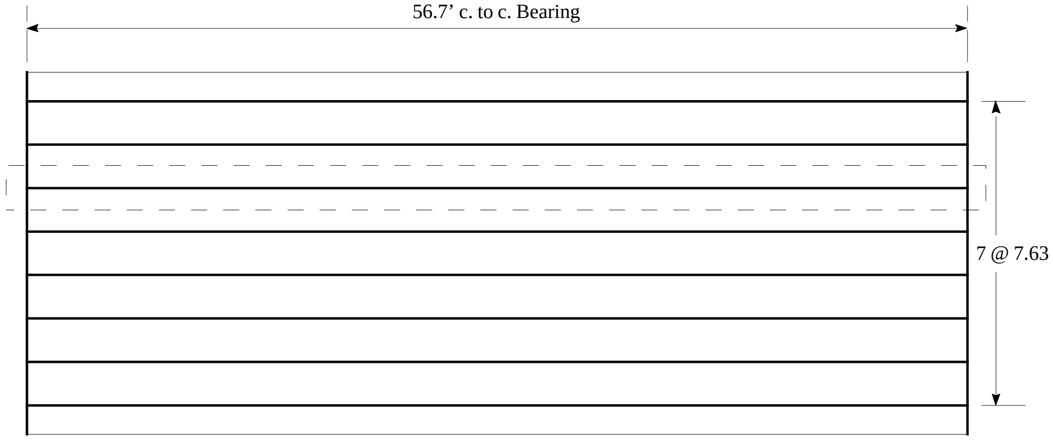


Figure 6.4: Plan of Highway Bridge

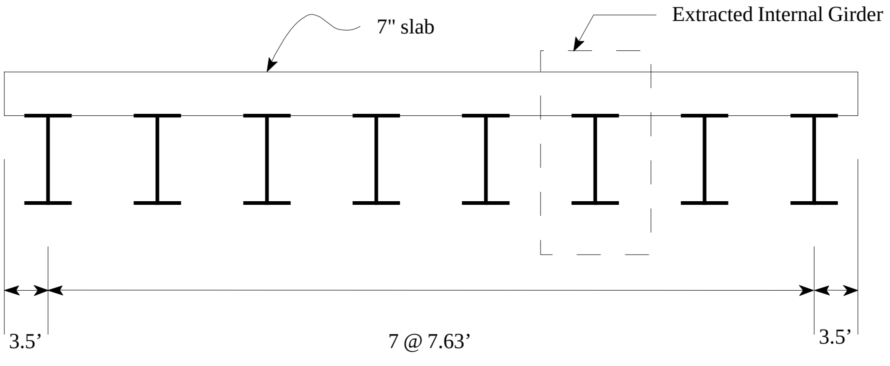


Figure 6.5: Typical Cross Section

The bridge girders are made of rolled beam W33x130 with a 14” × 3/4” steel cover plate. An elevation view of the bridge and the position of the steel cover plate is shown in Figure

6.6. The material properties are $F_y = 50$ ksi, and $E_s = 29000$ ksi. The effective cross sectional properties of the composite steel/concrete girder (with and without the cover plate) are computed with $n = E_s/E_c = 10$. The section properties are shown in Figure 6.7.

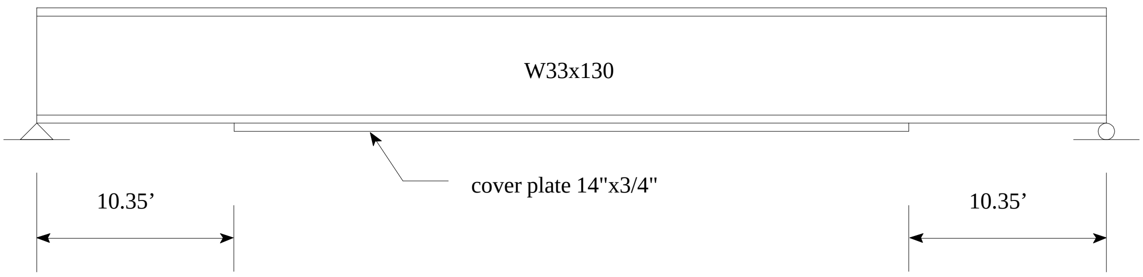


Figure 6.6: Elevation of Beam

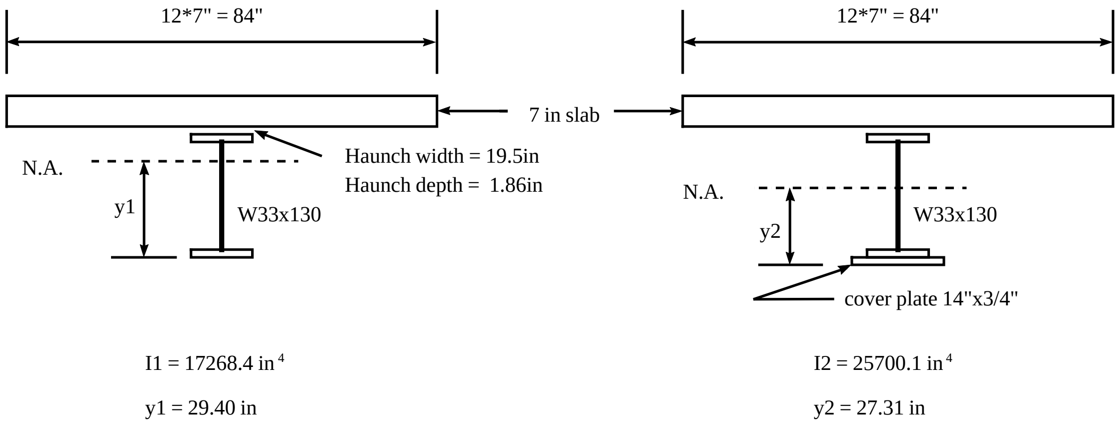


Figure 6.7: Section Properties (n = 10)

The bridge is subjected to dead and live external loadings. The design dead load includes 7 inches concrete slab, steel girder, and superimposed load. The design live load consists of a 72 kips HS-20 truck, which will be modeled as a single concentrated load moving load along the girder nodes.

**Input File :** The single-span bridge girder is modeled with 10 two-dimensional beam/column finite elements.

START OF INPUT FILE

```
/*
 *  ===============================================================
 *  Test input file for bridge design rule checking wsing 2D-beam
 *  element and static analysis
```

```
 *
 *  Written by:  Wane-Jang Lin                                June 1995
 *  ================================================================
 */

print "*** DEFINE PROBLEM SPECIFIC PARAMETERS \n\n";

NDimension         = 2;
NDofPerNode        = 3;
MaxNodesPerElement = 2;

StartMesh();
div_no = 10;

   print "*** GENERATE GRID OF NODES FOR FE MODEL \n\n";

   length = 56.70 ft;
   cov_dis = 10.35 ft;

   dL = length/div_no;
   x = 0 ft;  y = 0 ft;
   for( i=1; x<=length ; i=i+1 ) {
       AddNode(i, [x, y]);
       x = x+dL;
   }

   print "*** ATTACH ELEMENTS TO GRID OF NODES \n\n";

   elmt = 1;
   x = 0 ft;
   for( i=1; i<=div_no ; i=i+1 ) then {
      if( x<cov_dis || (x+dL)>(length-cov_dis) ) {
          AddElmt( elmt, [i, i+1], "girder1");
      } else {
          AddElmt( elmt, [i, i+1], "girder2");
      }
      elmt = elmt+1;
      x = x+dL;
   }

   print "*** DEFINE ELEMENT, SECTION AND MATERIAL PROPERTIES \n\n";

   I1 = 17268.4 in^4;  y1 = 29.40 in;  area1 = 38.3 in^2;
   I2 = 25700.1 in^4;  y2 = 27.31 in;  area1 = 48.8 in^2;

   ElementAttr("girder1")  { type     = "FRAME_2D";
                             section  = "no_cover";
                             material = "STEEL3";
                           }
   ElementAttr("girder2")  { type     = "FRAME_2D";
                             section  = "cover";
                             material = "STEEL3";
                           }
```

```
  SectionAttr("no_cover") { Izz     =   I1;
                            area    =   area1;
                            depth   =   33.09 in;
                            width   =   11.51 in;
                          }
  SectionAttr("cover")    { Izz     =   I2;
                            area    =   area2;
                            depth   =   33.09 in;
                            width   =   11.51 in;
                          }

/*
 *  ======================================
 *  Setup Boundary Conditions [dx, dy, rz]
 *  ======================================
 */

  FixNode(1, [1, 1, 0]);
  FixNode(div_no+1, [0, 1, 0]);

/*
 *  =====================================
 *  Compile and Print Finite Element Mesh
 *  =====================================
 */

  EndMesh();
  PrintMesh();
  SetUnitsType("US");
  stiff = Stiff();
  lu = Decompose(stiff);

/*
 *  =====================================
 *  Add Dead Nodal Loads [Fx, Fy, Mz]
 *  =====================================
 */

  weight = 0.199 kips/ft + 0.868 kips/ft;
  load = weight*dL;

  for( i=2; i<=div_no; i=i+1 ) {
       NodeLoad( i, [0 kips, -load, 0 kips*ft]);
  }
  NodeLoad(        1, [0 kips, -load/2, 0 kips*ft]);
  NodeLoad( div_no+1, [0 kips, -load/2, 0 kips*ft]);

  eload  = ExternalLoad();
  displ  = Substitution(lu, eload);
  max_displ_dead = GetDispl([div_no/2+1],displ);
  max_mom_dead   = GetStress([div_no/2],displ);
  max_sh_dead    = GetStress([1],displ);
  cover_mom_dead = GetStress([2],displ);
  PrintMatrix(max_displ_dead,max_mom_dead,max_sh_dead,cover_mom_dead);
```

```
/* Get Moment Diagram */

   moment_dead = Zero([ div_no+1 , 1 ]);
   for( i=1 ; i<=div_no ; i=i+1 ) {
        temp = GetStress([i],displ);
        if( i == 1 ) {  moment_dead[1][1] = temp[1][3];  }
        moment_dead[i+1][1] = temp[2][3];
   }
   PrintMatrix(moment_dead);

/* Zero-out dead loadings */

   for( i=2; i<=div_no; i=i+1 ) {
        NodeLoad( i, [0 kips, load, 0 kips*ft]);
   }
   NodeLoad(        1, [0 kips, load/2, 0 kips*ft]);
   NodeLoad( div_no+1, [0 kips, load/2, 0 kips*ft]);

/*
 *  =====================================
 *  Add Live Truck Load [Fx, Fy, Mz]
 *  =====================================
 */

   step   = div_no+1;
   influ_mid_mom = Zero([ step , 1 ]);
   influ_end_sh  = Zero([ step , 1 ]);
   envelop_mom_live  = Zero([ step , 1 ]);
   HS20   = 72 kips;
   mom_elmt = div_no/2;
   sh_elmt  = 1;

   for( i=1 ; i<=step ; i=i+1 ) {
       NodeLoad( i, [0 kips, -HS20, 0 kips*ft] );
       eload = ExternalLoad();
       displ  = Substitution( lu, eload );
       mid_mom = GetStress( [mom_elmt], displ );
       end_sh  = GetStress( [sh_elmt], displ );
       if( i == 1 ) then {
           mom_rang  = GetStress( [i], displ );
       } else {
           mom_rang  = GetStress( [i-1], displ );
       }

       /* Get moment influence line at the middle of span */

       influ_mid_mom[i][1] = mid_mom[2][3];

       /* Get Shear Influence line at the end support */

       influ_end_sh[i][1]  = end_sh[1][2];

       /* Create envelope of moment influence line at the middle of span */
```

```
      if( i == 1 ) then {
          envelop_mom_live[1][1]  = mom_rang[1][3];
      } else {
         envelop_mom_live[i][1]  = mom_rang[2][3];
      }

      if( i == mom_elmt+1 ) {
          max_displ_live = GetDispl([mom_elmt+1],displ);
          max_mom_live = GetStress([mom_elmt],displ);
      }
      if( i == sh_elmt ) {
          max_sh_live = GetStress([sh_elmt],displ);
          max_sh_live[1][2]  = max_sh_live[1][2] + HS20;
          influ_end_sh[1][1] = HS20;
      }
      if( i == 3 ) {
          cover_mom_live = GetStress([2],displ);
      }

      NodeLoad( i, [0 kips, HS20, 0 kips*ft] );
  }

  PrintMatrix(max_displ_live,max_mom_live,max_sh_live,cover_mom_live);
  PrintMatrix(influ_mid_mom, influ_end_sh, envelop_mom_live);

/*
 *  ============================
 *  Results for DL + (LL+Impact)
 *  ============================
 */

  impact = 1 + 50/(56.7+125);
  print "\nimpact factor = ",impact,"\n";
  max_displ  = max_displ_dead + impact*max_displ_live;
  max_mom    = max_mom_dead + impact*max_mom_live;
  max_sh     = max_sh_dead + impact*max_sh_live;
  cover_mom  = cover_mom_dead + impact*cover_mom_live;

  PrintMatrix(max_displ,max_mom,max_sh,cover_mom);

/*
 *  =========================================================
 *  WSD Code Checking for Deflections and Stress Requirements
 *  =========================================================
 */

  print "\n\nSTART ASD CODE CHECKING::\n";

/* Deflection Checking */

  if( -impact*max_displ_live[1][2] > (1/800)*length ) then {
      print "\n\tWarning: (LL+I) deflection exceeds 1/800 span\n";
  } else {
```

```
        print "\n\tOK : (LL+I) deflection less than 1/800 span\n";
    }

/* Moment Stress Checking */

    my_material = GetMaterial([1]);
    my_section  = GetSection([1]);

    /* max stress without cover plate */
    stress1 = cover_mom[2][3]*y1/I1;

    /* max stress with cover plate (in the middle of span) */
    stress2 = max_mom[2][3]*y2/I2;

    if( stress1 > 0.55*my_material[3][1] ) then{
        print "\n\tWarning : moment stress without cover plate larger than 0.55*Fy\n";
    } else {
        if( stress2 > 0.55*my_material[3][1] ) then{
            print "\n\tWarning : moment stress with cover plate larger than 0.55*Fy\n";
        } else {
            print "\n\tOK : moment stress less than 0.55*Fy\n";
        }
    }

/* Shear Stress Checking */

    shear = max_sh[1][2]/my_section[11][1];
    if( shear > 0.33*my_material[3][1] ) then{
        print "\n\tWarning : shear stress larger than 0.33*Fy\n";
    } else {
        print "\n\tOK : shear stress less than 0.33*Fy\n";
    }

    quit;
```

---

Points to note are:

[1] In this example, we check the analysis result with AASHTO WSD specification. The impact factor for live load is based on the formula AASHTO Eq. (3-1)

$$I = \frac{50}{L + 125}$$

in which
$I$ = impact fraction (maximum 30 percent);
$L$ = length in feet of the portion of the span;

[2] The deflection checking is based on AASHTO Art.10.6.2, the deflection due to service live load plus impact shall not exceed 1/800 of the span.

[3] The allowable stress is $0.55 \times F_y$ for tension and compression member, $0.33 \times F_y$ for shear in web.

**Abbreviated Output File :** The opening sections of the following output file contain details of the 11 node, 10 element mesh. The left-hand side of the girder is modeled as a pin fixed against translation. The right-hand side of the girder is supported on a roller.

The latter components of the output file contain summaries of the shear and moment envelopes, together with maximum values of response quantities.

START OF ABBREVIATED OUTPUT FILE

```
===========================================================================
Title : DESCRIPTION OF FINITE ELEMENT MESH
===========================================================================

 Problem_Type:  Static Analysis

=======================
Profile of Problem Size
=======================

   Dimension of Problem        =      2

   Number Nodes                =     11
   Degrees of Freedom per node =      3
   Max No Nodes Per Element    =      2

   Number Elements             =     10
   Number Element Attributes   =      2
   Number Loaded Nodes         =      0
   Number Loaded Elements      =      0


------------------------------------------------------------------------------
Node#      X_coord           Y_coord          Tx     Ty     Rz
------------------------------------------------------------------------------


    1    0.0000e+00 ft    0.0000e+00 ft    -1    -2     1
    2    5.6700e+00 ft    0.0000e+00 ft     2     3     4
    3    1.1340e+01 ft    0.0000e+00 ft     5     6     7
    4    1.7010e+01 ft    0.0000e+00 ft     8     9    10
    5    2.2680e+01 ft    0.0000e+00 ft    11    12    13
    6    2.8350e+01 ft    0.0000e+00 ft    14    15    16
    7    3.4020e+01 ft    0.0000e+00 ft    17    18    19
    8    3.9690e+01 ft    0.0000e+00 ft    20    21    22
    9    4.5360e+01 ft    0.0000e+00 ft    23    24    25
   10    5.1030e+01 ft    0.0000e+00 ft    26    27    28
   11    5.6700e+01 ft    0.0000e+00 ft    29    -3    30
```

```
---------------------------------------------------------------------
Element#     Type         node[1]   node[2]      Element_Attr_Name
-----------------------------------------------------------------------

      1   FRAME_2D            1         2      girder1
      2   FRAME_2D            2         3      girder1
      3   FRAME_2D            3         4      girder2
      4   FRAME_2D            4         5      girder2
      5   FRAME_2D            5         6      girder2
      6   FRAME_2D            6         7      girder2
      7   FRAME_2D            7         8      girder2
      8   FRAME_2D            8         9      girder2
      9   FRAME_2D            9        10      girder1
     10   FRAME_2D           10        11      girder1

---------------------
Element Attribute Data :
---------------------


ELEMENT_ATTR No.   1  : name = "girder1"
                      : section = "no_cover"
                      : material = "steel"
                      : type = FRAME_2D
                      : gdof [0] =    1 : gdof[1] =    2 : gdof[2] =    3
                      : Young's Modulus =  E =        2.900e+04 ksi
                      : Yielding Stress = fy =        5.000e+01 ksi
                      : Poisson's ratio = nu =        3.000e-01
                      : Density         =        0.000e+00 (null)
                      : Inertia Izz     =        1.727e+04 in^4
                      : Area            =        3.809e+02 in^2


ELEMENT_ATTR No.   2  : name = "girder2"
                      : section = "cover"
                      : material = "steel"
                      : type = FRAME_2D
                      : gdof [0] =    1 : gdof[1] =    2 : gdof[2] =    3
                      : Young's Modulus =  E =        2.900e+04 ksi
                      : Yielding Stress = fy =        5.000e+01 ksi
                      : Poisson's ratio = nu =        3.000e-01
                      : Density         =        0.000e+00 (null)
                      : Inertia Izz     =        2.570e+04 in^4
                      : Area            =        3.809e+02 in^2

============= End of Finite Element Mesh Description ==============


MATRIX : "max_displ_dead"

row/col                  1            2            3
        units           in           in          rad
   1           0.00000e+00 -3.44203e-01  5.69206e-18
```

```
MATRIX : "max_mom_dead"

row/col                 1           2           3
       units          lbf         lbf      lbf.in
   1          -0.00000e+00  3.02494e+03 -4.93961e+06
   2           0.00000e+00 -3.02494e+03  5.14543e+06

MATRIX : "max_sh_dead"

row/col                 1           2           3
       units          lbf         lbf      lbf.in
   1          -0.00000e+00  2.72245e+04 -8.24290e-09
   2           0.00000e+00 -2.72245e+04  1.85236e+06

MATRIX : "cover_mom_dead"

row/col                 1           2           3
       units          lbf         lbf      lbf.in
   1          -0.00000e+00  2.11746e+04 -1.85236e+06
   2           0.00000e+00 -2.11746e+04  3.29308e+06

MATRIX : "moment_dead"

row/col                 1
       units
   1   lbf.in  -8.24290e-09
   2   lbf.in   1.85236e+06
   3   lbf.in   3.29308e+06
   4   lbf.in   4.32216e+06
   5   lbf.in   4.93961e+06
   6   lbf.in   5.14543e+06
   7   lbf.in   4.93961e+06
   8   lbf.in   4.32216e+06
   9   lbf.in   3.29308e+06
  10   lbf.in   1.85236e+06
  11   lbf.in   8.24290e-09

MATRIX : "max_displ_live"

row/col                 1           2           3
       units           in          in         rad
   1           0.00000e+00 -6.53755e-01  1.06252e-17

MATRIX : "max_mom_live"

row/col                 1           2           3
       units          lbf         lbf      lbf.in
   1          -0.00000e+00  3.60000e+04 -9.79776e+06
   2           0.00000e+00 -3.60000e+04  1.22472e+07

MATRIX : "max_sh_live"

row/col                 1           2           3
       units          lbf         lbf      lbf.in
```

```
   1           -0.00000e+00  7.20000e+04  0.00000e+00
   2            0.00000e+00 -0.00000e+00  0.00000e+00

MATRIX : "cover_mom_live"

row/col                  1            2            3
        units          lbf          lbf       lbf.in
   1           -0.00000e+00  5.76000e+04 -3.91910e+06
   2            0.00000e+00 -5.76000e+04  7.83821e+06

MATRIX : "influ_mid_mom"

row/col                  1
        units
   1   lbf.in   0.00000e+00
   2   lbf.in   2.44944e+06
   3   lbf.in   4.89888e+06
   4   lbf.in   7.34832e+06
   5   lbf.in   9.79776e+06
   6   lbf.in   1.22472e+07
   7   lbf.in   9.79776e+06
   8   lbf.in   7.34832e+06
   9   lbf.in   4.89888e+06
  10   lbf.in   2.44944e+06
  11   lbf.in   0.00000e+00

MATRIX : "influ_end_sh"

row/col                  1
        units
   1      lbf   7.20000e+04
   2      lbf   6.48000e+04
   3      lbf   5.76000e+04
   4      lbf   5.04000e+04
   5      lbf   4.32000e+04
   6      lbf   3.60000e+04
   7      lbf   2.88000e+04
   8      lbf   2.16000e+04
   9      lbf   1.44000e+04
  10      lbf   7.20000e+03
  11      lbf   0.00000e+00

MATRIX : "envelop_mom_live"

row/col                  1
        units
   1   lbf.in   0.00000e+00
   2   lbf.in   4.40899e+06
   3   lbf.in   7.83821e+06
   4   lbf.in   1.02876e+07
   5   lbf.in   1.17573e+07
   6   lbf.in   1.22472e+07
   7   lbf.in   1.17573e+07
   8   lbf.in   1.02876e+07
```

```
   9   lbf.in   7.83821e+06
  10   lbf.in   4.40899e+06
  11   lbf.in   0.00000e+00

impact factor =      1.275

MATRIX : "max_displ"

row/col                 1            2            3
        units          in           in          rad
   1            0.00000e+00 -1.17786e+00  1.92411e-17

MATRIX : "max_mom"

row/col                 1            2            3
        units         lbf          lbf       lbf.in
   1           -0.00000e+00  4.89314e+04 -1.74335e+07
   2            0.00000e+00 -4.89314e+04  2.07628e+07

MATRIX : "max_sh"

row/col                 1            2            3
        units         lbf          lbf       lbf.in
   1           -0.00000e+00  1.19037e+05 -8.24290e-09
   2            0.00000e+00 -2.72245e+04  1.85236e+06

MATRIX : "cover_mom"

row/col                 1            2            3
        units         lbf          lbf       lbf.in
   1           -0.00000e+00  9.46249e+04 -6.84991e+06
   2            0.00000e+00 -9.46249e+04  1.32882e+07


START ASD CODE CHECKING::

   OK : (LL+I) deflection less than 1/800 span

   OK : moment stress less than 0.55*Fy

   OK : shear stress less than 0.33*Fy
```

---

A summary of the bridge response is contained in Figures 6.8 to 6.10. The following points are noted:

[1] Since this is a simple-supported bridge, the maximum displacement and maximum bending moment will occur at the middle of the span. The maximum shear force will occur at the end support.

[2] The final results of moment, shear and displacement are calculated according to AASHTO ASD request: Total = DL + impact * LL.

[3] The stress due to bending is given by:

$$\text{Moment stress} = \frac{M \cdot y}{I}. \tag{6.1}$$

Because there are two different section properties, we have to calculate not only the maximum moment stress in the middle of the span (stress2), but also the moment stress of where the section changed (stress1).

$$\text{Shear stress} = \frac{V}{tw \cdot d}. \tag{6.2}$$

We assume that the shear force is carried by the girder web alone, and therefore, we only check for maximum shear at the end support.

[4] The influence line for the bending moment at the middle of the span – details are shown in Figure: 6.9 – is obtained by iteratively positioning one truck load at a finite element node, then solving for the reaction forces. A similar procedure is employed to compute the influence line of shear force at the end support – see Figure 6.10.

[5] Figure 6.8) shows the distribution of bending moments due to truck loadings (we note in passing that the bending momend diagram corresponds to an envelope of the moment influence lines of the truck load).

[6] The follow array elements are used in the generation of program output:

```
max_displ[1][2] = the maximum displacement of the beam.
max_mom[2][3]   = the maximum moment.
max_sh[1][2]    = the maximum shear force.
cover_mom[2][3] = the moment at where the bridge section changed.
```

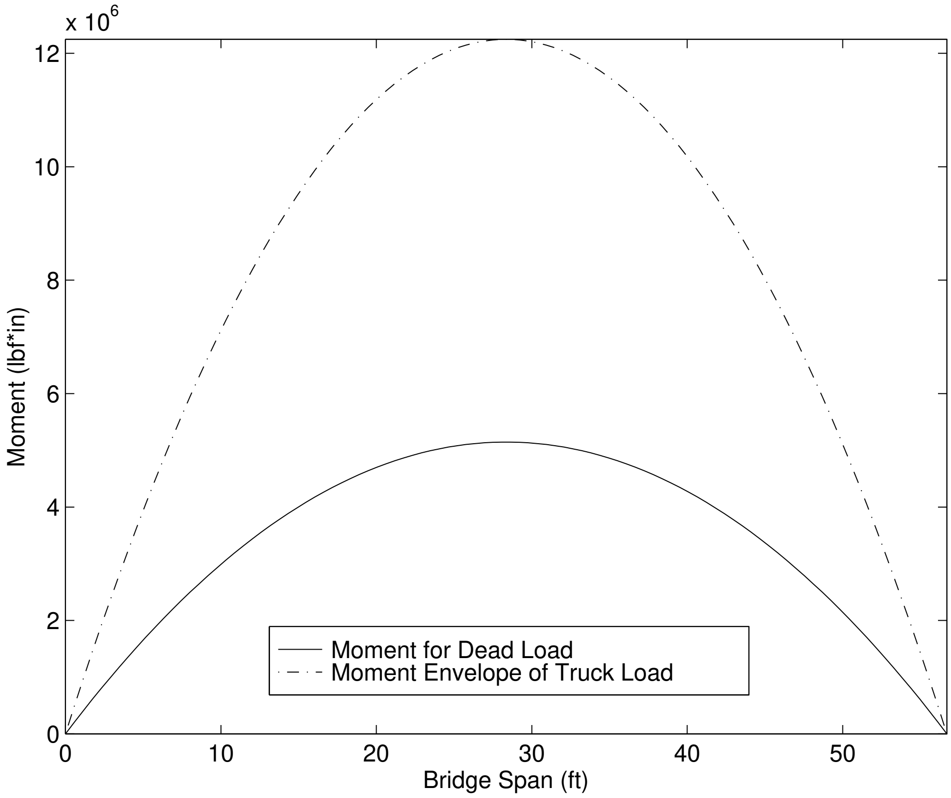


Figure 6.8: Moment Diagram of Dead Load and Truck Load

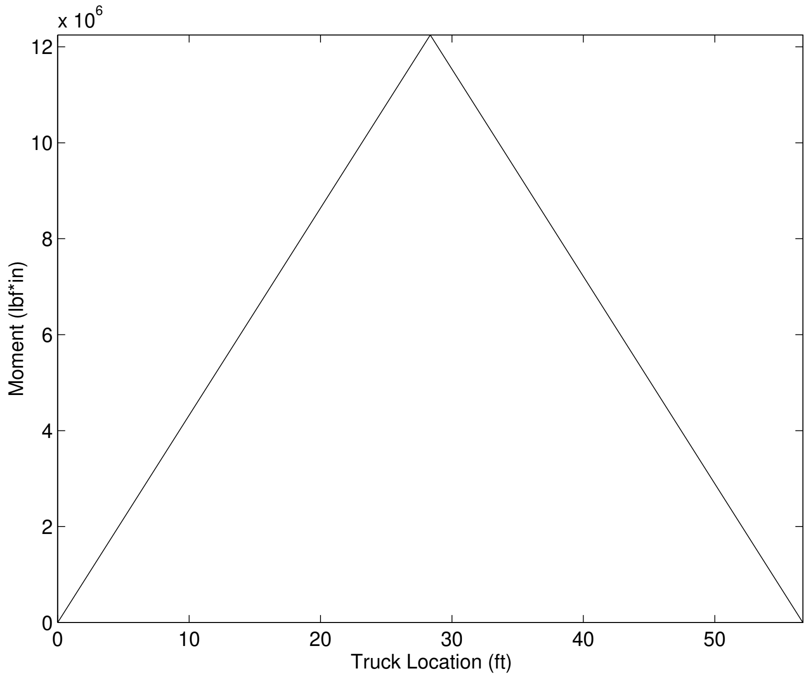


Figure 6.9: Influence Line of Moment at Mid-Span

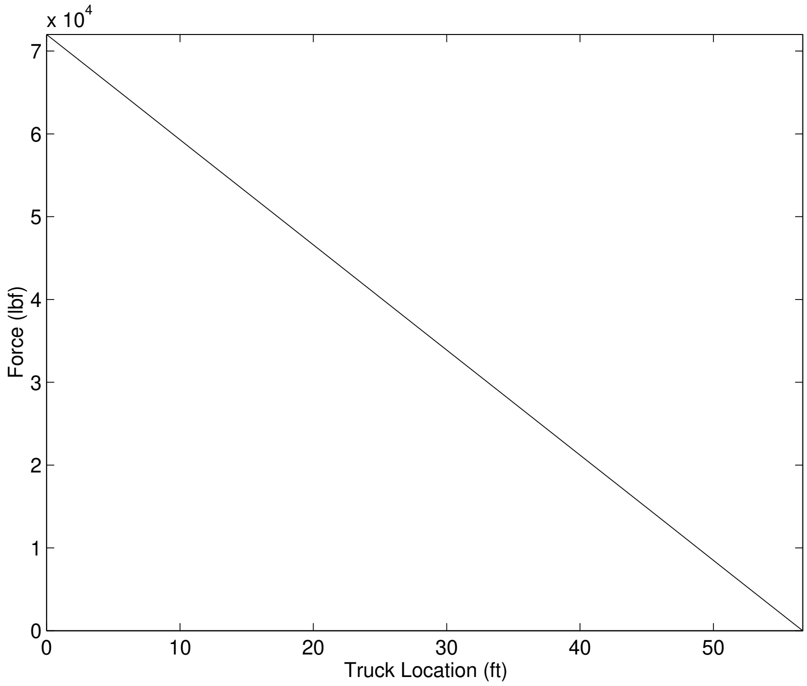


Figure 6.10: Influence Line of Shear at End Support

### 6.1.3 Three-Dimensional Analysis of Highway Bridge

**Description of Bridge Problem :** In this section we will conduct a two-part analysis of a three-dimensional two-span highway bridge structure.

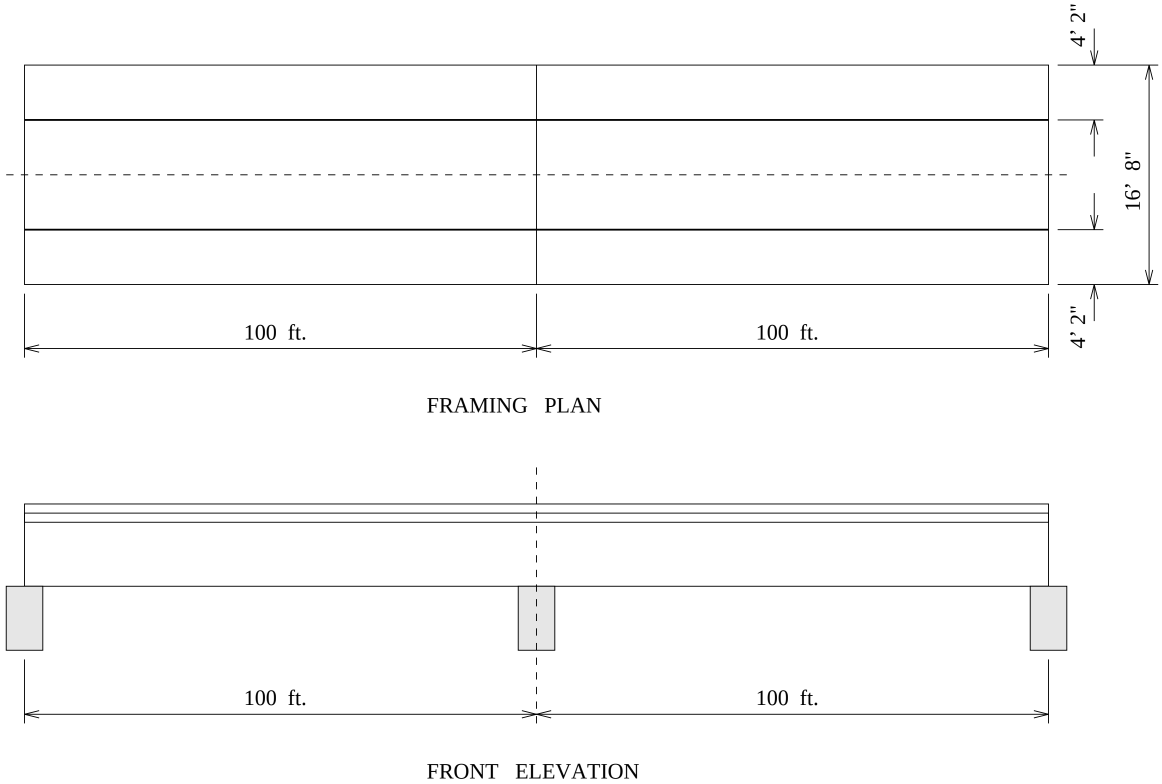


Figure 6.11: Highway Bridge : Plan and Front Elevation of Two-Span Continuous Highway Bridge

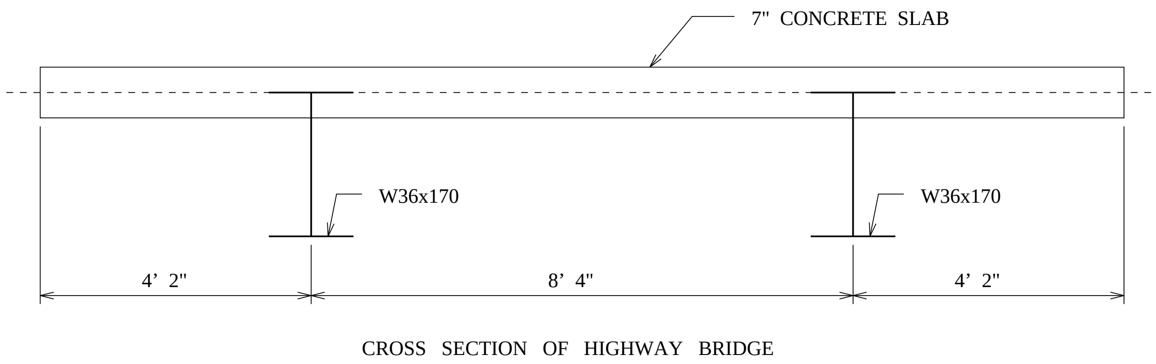


Figure 6.12: Highway Bridge : Cross Section of Two-Span Continuous Highway Bridge

A plan and front elevation view of the highway bridge is shown in Figure 6.11. The bridge

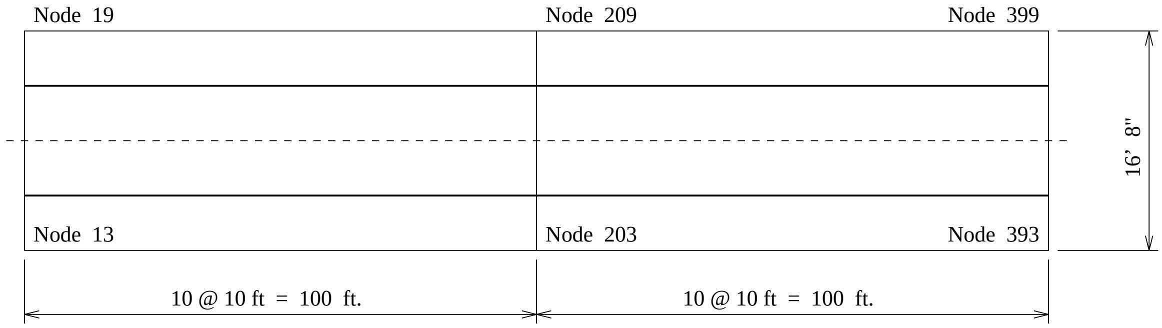


Figure 6.13: Highway Bridge : Plan of Finite Element Mesh for Two-Span Continuous Highway Bridge

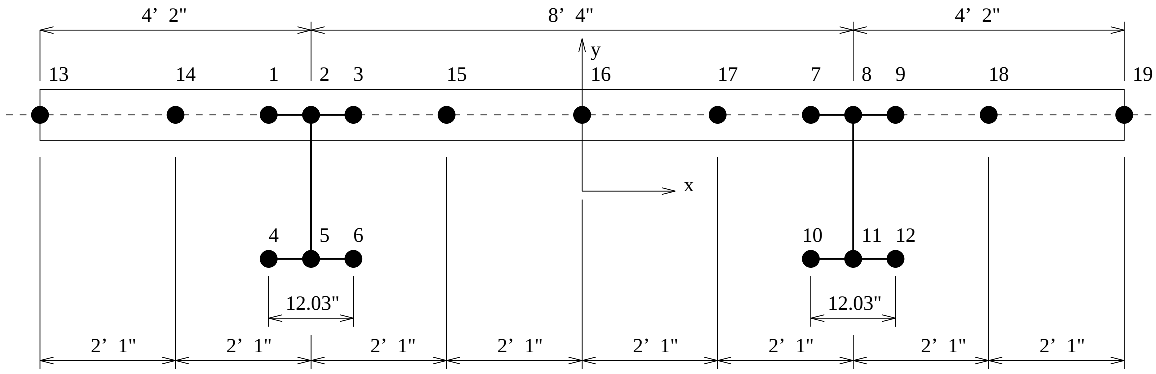


Figure 6.14: Highway Bridge : Cross Section of Finite Element Mesh for Two-Span Continuous Highway Bridge

has two spans, each 100 ft long. The width of the bridge is 16 ft - 8 inches. A cross section view of the bridge is shown in Figure 6.12.

The bridge is constructed from one steel and one concrete material type. The structural steel has $F_y$ = 50 ksi, Poisson's ratio $v$ = 0.3, and E = 29,000 ksi. For the concrete slab, $f'_c$ = 4000 psi, Poisson's ratio $v$ = 0.3, $E_c$ = 3625 ksi, and unit weight = 150 pcf. The W36x170 steel section has the following properties; W36x170 : d = 36.17 in, $b_f$ = 12.03 in, $t_f$ = 1.1 in, $t_w$ = 0.68 in, and weight = 170 lbf/ft. The concrete slab has thickness 7 in.

The highway bridge will be analyzed for two loading conditions. First, we compute the deflections of the bridge due to gravity loads alone. The concrete slab weighs $150pcf \cdot 7in = 87.5lbf/ft^2$. The girder weight is $170lbf/ft$. In part two of the bridge analysis, a 1000 kip concentrated live load moves along one of the outer bridge girders. We compute and plot the influence line for the moving load.

The boundary conditions for the left-hand sidee of the bridge are a hinged support.

The right-hand side of the bridge is supported on a roller. The finite element model has 399 nodes and 440 shell elements. After the boundary conditions are applied, the model has 2374 d.o.f. The bridge is subject to a 1000 kip concentrated moving live load.

**Input File :** The following input file defines the bridge geometry, boundary conditions, finite element, section and material types, and the external loads due to the dead weight of the bridge.

START OF INPUT FILE

```
/*
 *  ===================================================================
 *  Analysis of Two Span Continuous Composite Beam (100-100ft) assuming
 *  neutral axis of concrete deck coincide with top girder flange
 *
 *  Written By: Wane-Jang Lin                              December, 1994
 *  ===================================================================
 */

/* [a] : Setup problem specific parameters */

   print "*** DEFINE PROBLEM SPECIFIC PARAMETERS \n\n";

   NDimension         = 3;
   NDofPerNode        = 6;
   MaxNodesPerElement = 4;

   StartMesh();

   print "*** GENERATE GRID OF NODES FOR FE MODEL \n\n";

   girder   = 2;
   span     = 100 ft;
   spacing  = 8 ft + 4 in;
   slab     = 7 in;
   height   = 36.17 in - 1.1 in;
   flange   = 12.03 in;

   div_L    = 10;
   div_S    = 4;
   delta_L  = span/div_L;
   delta_S  = spacing/div_S;
   delta_f  = flange/2;
   delta_h  = height/2;

   section_no = 2*div_L + 1;
   nodes_per_girder = 6;
   nodes_per_section = nodes_per_girder*girder + (2*div_S+1) - girder;

   node = 0;
   x = 0 in;
   for(i=1 ; i<=section_no ; i=i+1 ) {
       x  =  delta_L*(i-1);
```

```
    for( j=1 ; j<=girder ; j=j+1 ) {
         y    = -spacing/2 + spacing*(j-1);
         AddNode( node+1, [ x, (y-delta_f),  delta_h ] );
         AddNode( node+2, [ x,          y ,  delta_h ] );
         AddNode( node+3, [ x, (y+delta_f),  delta_h ] );
         AddNode( node+4, [ x, (y-delta_f), -delta_h ] );
         AddNode( node+5, [ x,          y , -delta_h ] );
         AddNode( node+6, [ x, (y+delta_f), -delta_h ] );
         node = node + nodes_per_girder;
    }

    yy = spacing/2;
    for( j=1 ; j<=(2*div_S+1) ; j=j+1 ) {
         y = -spacing + delta_S*(j-1);
         if( (y!=yy) && (y!=-yy) ) {
              node = node + 1;
              AddNode( node, [ x, y, delta_h ] );
         }
    }
}

print "*** ATTACH ELEMENTS TO GRID OF NODES \n\n";

elmtno = 0;
a = 0;
b = 0 + nodes_per_section;

for( i=1 ; i<section_no ; i=i+1 ) {
for( j=1 ; j<=girder ; j=j+1 ) {
    AddElmt( elmtno+1, [ a+1, b+1, b+2, a+2 ],"girder_flange_attr" );
    AddElmt( elmtno+2, [ a+2, b+2, b+3, a+3 ],"girder_flange_attr" );
    AddElmt( elmtno+3, [ a+2, b+2, b+5, a+5 ],"girder_web_attr" );
    AddElmt( elmtno+4, [ a+4, b+4, b+5, a+5 ],"girder_flange_attr" );
    AddElmt( elmtno+5, [ a+5, b+5, b+6, a+6 ],"girder_flange_attr" );
    elmtno = elmtno + nodes_per_girder - 1;
    a = a + nodes_per_girder;
    b = b + nodes_per_girder;
}
a = a + 1;
b = b + 1;
for( j=1 ; j<=girder ; j=j+1 ) {
    AddElmt( elmtno+1, [   a,   b, b+1, a+1 ], "deck_attr" );
    c = b - (girder+1-j)*nodes_per_girder - 3*(j-1);
    d = a - (girder+1-j)*nodes_per_girder - 3*(j-1);
    AddElmt( elmtno+2, [ a+1, b+1,   c,   d ], "deck_attr" );
    a = d;
    b = c;
    AddElmt( elmtno+3, [   a,   b, b+1, a+1 ], "deck_attr" );
    AddElmt( elmtno+4, [ a+1, b+1, b+2, a+2 ], "deck_attr" );
    a = a + 2;
    b = b + 2;
    c = c + (girder+1-j)*nodes_per_girder + (3*j-2) + 1;
    d = d + (girder+1-j)*nodes_per_girder + (3*j-2) + 1;
    AddElmt( elmtno+5, [   a,   b,   c,   d ], "deck_attr" );
```

```
        AddElmt( elmtno+6, [   d,   c, c+1, d+1 ], "deck_attr" );
        elmtno = elmtno + 6;
        a = d + 1;
        b = c + 1;
    }
    }

    print "*** DEFINE ELEMENT, SECTION AND MATERIAL PROPERTIES \n\n";

    ElementAttr("girder_flange_attr") { type  = "SHELL_4NQ";
                                        section  = "girder_flange";
                                        material = "STEEL3";
                                      }

    ElementAttr("girder_web_attr") { type     = "SHELL_4NQ";
                                        section  = "girder_web";
                                        material = "STEEL3";
                                      }

    ElementAttr("deck_attr") {    type = "SHELL_4NQ";
                               section = "deck";
                              material = "concrete";
                             }

    SectionAttr("girder_flange")      { thickness = 1.100 in; }
    SectionAttr("girder_web")         { thickness = 0.680 in; }
    SectionAttr("deck")        { thickness = 7 in; }
    MaterialAttr("concrete")   { poisson   = 0.3;
                                 yield     = 0.85*(4000 psi);
                                 E         = (29000 ksi)/8;
                               }

    print "*** SET UP BOUNDARY CONDITIONS \n\n";

    bc_hs = [ 1, 1, 1, 1, 0, 0 ]; /* hinged support  */
    bc_rs = [ 0, 1, 1, 1, 0, 0 ]; /* roller support  */

    for( i=1 ; i<=girder ; i=i+1 ) {
         node = nodes_per_girder*(i-1) + 5;
         FixNode(node, bc_hs);
         node = node + div_L*nodes_per_section;
         FixNode(node, bc_rs);
         node = node + div_L*nodes_per_section;
         FixNode(node, bc_rs);
    }

/* [b] : Apply Point Nodal Loads */

    print "*** SET UP LOADS \n\n";

    slab_load     = (150 lbf/ft^3)*slab;
    girder_weight = 170 lbf/ft;

    Fx = 0 lbf;    Fy = 0 lbf;
```

```
Mx = 0 lbf*in; My = 0 lbf*in; Mz = 0 lbf*in;

/* [b.1] : load for corner nodes of deck  */

   Fz = -slab_load*delta_L/2*delta_S/2;
   nodal_load = [Fx, Fy, Fz, Mx, My, Mz];
   node = nodes_per_girder*girder + 1;
   NodeLoad(node, nodal_load);
   node = node + nodes_per_section*(section_no-1);
   NodeLoad(node, nodal_load);
   node = node + 2*div_S - girder;
   NodeLoad(node, nodal_load);
   node = node - nodes_per_section*(section_no-1);
   NodeLoad(node, nodal_load);

/* [b.2] load for edge nodes along x-direction  */

   Fz = -slab_load*delta_L*delta_S/2;
   nodal_load = [Fx, Fy, Fz, Mx, My, Mz];
   node = nodes_per_section + nodes_per_girder*girder + 1;
   for( i=2 ; i<section_no ; i=i+1 ) {
        NodeLoad(node, nodal_load);
        NodeLoad((node+2*div_S-girder), nodal_load);
        node = node + nodes_per_section;
   }

/* [b.3] load for edge nodes along y-direction  */

/* node 16 */
   Fz = -slab_load*delta_L/2*delta_S;
   nodal_load = [Fx, Fy, Fz, Mx, My, Mz];
   node = nodes_per_girder*girder + div_S;
   NodeLoad(node, nodal_load);
   node = node + nodes_per_section*(section_no-1);
   NodeLoad(node, nodal_load);

/* node 14, 15, 17, 18 */
   Fz = -slab_load*delta_L/2*(delta_S-delta_f/2);
   nodal_load = [Fx, Fy, Fz, Mx, My, Mz];
   node = nodes_per_girder*girder + 1;
   for( i=1 ; i<=2 ; i=i+1 ) {
        NodeLoad(node+1, nodal_load);
        NodeLoad(node+2, nodal_load);
        NodeLoad(node+4, nodal_load);
        NodeLoad(node+5, nodal_load);
        node = node + nodes_per_section*(section_no-1);
   }

/* node 1, 3, 7, 9 */
   Fz = -slab_load*delta_L/2*delta_S/2;
   nodal_load = [Fx, Fy, Fz, Mx, My, Mz];
   node = 0;
   for( i=1 ; i<=2 ; i=i+1 ) {
        NodeLoad(node+1, nodal_load);
```

```
        NodeLoad(node+3, nodal_load);
        NodeLoad(node+7, nodal_load);
        NodeLoad(node+9, nodal_load);
        node = node + nodes_per_section*(section_no-1);
   }

/* node 2, 8 */
   Fz = -slab_load*delta_L/2*delta_f;
   nodal_load = [Fx, Fy, Fz, Mx, My, Mz];
   node = 0;
   for( i=1 ; i<=2 ; i=i+1 ) {
        NodeLoad(node+2, nodal_load);
        NodeLoad(node+8, nodal_load);
        node = node + nodes_per_section*(section_no-1);
   }

/* [b.4] load for middle nodes */

/* node 35... */
   Fz = -slab_load*delta_L*delta_S;
   nodal_load = [Fx, Fy, Fz, Mx, My, Mz];
   node = nodes_per_girder*girder + div_S;
   for( i=2 ; i<section_no ; i=i+1 ) {
        node = node + nodes_per_section;
        NodeLoad(node, nodal_load);
   }

/* node 33, 34, 36, 37... */
   Fz = -slab_load*delta_L*(delta_S-delta_f/2);
   nodal_load = [Fx, Fy, Fz, Mx, My, Mz];
   node = nodes_per_girder*girder + 1;
   for( i=2 ; i<section_no ; i=i+1 ) {
        node = node + nodes_per_section;
        NodeLoad(node+1, nodal_load);
        NodeLoad(node+2, nodal_load);
        NodeLoad(node+4, nodal_load);
        NodeLoad(node+5, nodal_load);
   }

/* node 20, 22, 26, 28... */
   Fz = -slab_load*delta_L*delta_S/2;
   nodal_load = [Fx, Fy, Fz, Mx, My, Mz];
   node = 0;
   for( i=2 ; i<section_no ; i=i+1 ) {
        node = node + nodes_per_section;
        NodeLoad(node+1, nodal_load);
        NodeLoad(node+3, nodal_load);
        NodeLoad(node+7, nodal_load);
        NodeLoad(node+9, nodal_load);
   }

/* node 21, 27... */
   Fz = -slab_load*delta_L*delta_f;
   nodal_load = [Fx, Fy, Fz, Mx, My, Mz];
```

```
      node = 0;
      for( i=2 ; i<section_no ; i=i+1 ) {
           node = node + nodes_per_section;
           NodeLoad(node+2, nodal_load);
           NodeLoad(node+8, nodal_load);
      }

   /* [b.5] load for girder weight */

   /* end node 2, 8 */
      Fz = -girder_weight*delta_L/2;
      nodal_load = [Fx, Fy, Fz, Mx, My, Mz];
      node = 0;
      for( i=1 ; i<=2 ; i=i+1 ) {
           NodeLoad(node+2, nodal_load);
           NodeLoad(node+8, nodal_load);
           node = node + nodes_per_section*(section_no-1);
      }

   /* middle node 21, 27... */
      Fz = -girder_weight*delta_L;
      nodal_load = [Fx, Fy, Fz, Mx, My, Mz];
      node = 0;
      for( i=2 ; i<section_no ; i=i+1 ) {
           node = node + nodes_per_section;
           NodeLoad(node+2, nodal_load);
           NodeLoad(node+8, nodal_load);
      }

/* [c] : Compile and Print Finite Element Mesh */

   EndMesh();
   PrintMesh();

/* [d] : Compute Stiffness Matrix and External Load Vector */

   print "\n";
   print "*** COMPUTE AND PRINT STIFFNESS AND EXTERNAL LOAD MATRICES \n\n";

   SetUnitsType("US");

   eload = ExternalLoad();
   stiff = Stiff();

/* [e] : Compute and print static displacements */

   print "\n*** STATIC ANALYSIS PROBLEM \n\n";

   displ  = Solve( stiff, eload);
   PrintDispl(displ);
   PrintStress(displ);

   quit;
```

**Abbreviated Output File :** Several thousand lines of output are generated by ALADDIN and the abovementioned input file defined. Hence, we present an abbreviated summary of the output together with contour and three-dimensional views of the bridge deck deflections.

START OF ABBREVIATED OUTPUT FILE

```
============================================================================
Title : DESCRIPTION OF FINITE ELEMENT MESH
============================================================================

 Problem_Type:  Static Analysis

=======================
Profile of Problem Size
=======================

   Dimension of Problem        =      3

   Number Nodes                =    399
   Degrees of Freedom per node =      6
   Max No Nodes Per Element    =      4

   Number Elements             =    440
   Number Element Attributes   =      3
   Number Loaded Nodes         =    315
   Number Loaded Elements      =      0

----------------------------------------------------------------------------------
Node#       X_coord       Y_coord       Z_coord    Dx    Dy    Dz    Rx    Ry    Rz
----------------------------------------------------------------------------------

    1  0.000e+00 ft -4.667e+00 ft  1.753e+01 in     1     2     3     4     5     6
    2  0.000e+00 ft -4.166e+00 ft  1.753e+01 in     7     8     9    10    11    12
    3  0.000e+00 ft -3.665e+00 ft  1.753e+01 in    13    14    15    16    17    18
    4  0.000e+00 ft -4.667e+00 ft -1.753e+01 in    19    20    21    22    23    24

    ....... details of nodal coordinates deleted .......

  398  2.000e+02 ft  6.250e+00 ft  1.753e+01 in  2363  2364  2365  2366  2367  2368
  399  2.000e+02 ft  8.333e+00 ft  1.753e+01 in  2369  2370  2371  2372  2373  2374

---------------------------------------------------------------------------------
Element#     Type      node[1]   node[2]   node[3]   node[4]    Element_Attr_Name
---------------------------------------------------------------------------------

       1   SHELL_4NQ         1        20        21         2    girder_flange_attr
       2   SHELL_4NQ         2        21        22         3    girder_flange_attr
       3   SHELL_4NQ         2        21        24         5    girder_web_attr
       4   SHELL_4NQ         4        23        24         5    girder_flange_attr
```

```
   ....... details of shell finite elements removed .......

     437   SHELL_4NQ          368       387       388       369      deck_attr
     438   SHELL_4NQ          369       388       389       370      deck_attr
     439   SHELL_4NQ          370       389       398       379      deck_attr
     440   SHELL_4NQ          379       398       399       380      deck_attr

---------------------
Element Attribute Data :
---------------------

ELEMENT_ATTR No.   1  : name = "girder_flange_attr"
                      : section = "girder_flange"
                      : material = "STEEL3"
                      : type = SHELL_4NQ
                      : dof-mapping : gdof[0] =  1 : gdof[1] =    2 : gdof[2] =  3
                                      gdof[3] =  4 : gdof[4] =    5 : gdof[5] =  6
                      : Young's Modulus =  E =        2.900e+04 ksi
                      : Yielding Stress = fy =        5.000e+01 ksi
                      : Poisson's ratio = nu =        3.000e-01
                      : Density         =       0.000e+00 (null)
                      : Inertia Izz     =       0.000e+00 (null)
                      : Area            =       0.000e+00 (null)

ELEMENT_ATTR No.   2  : name = "girder_web_attr"
                      : section = "girder_flange"
                      : material = "STEEL3"
                      : type = SHELL_4NQ
                      : dof-mapping : gdof[0] =  1 : gdof[1] =  2 : gdof[2] =  3
                                      gdof[3] =  4 : gdof[4] =  5 : gdof[5] =  6
                      : Young's Modulus =  E =        2.900e+04 ksi
                      : Yielding Stress = fy =        5.000e+01 ksi
                      : Poisson's ratio = nu =        3.000e-01
                      : Density         =       0.000e+00 (null)
                      : Inertia Izz     =       0.000e+00 (null)
                      : Area            =       0.000e+00 (null)


ELEMENT_ATTR No.   3  : name = "deck_attr"
                      : section = "deck"
                      : material = "concrete"
                      : type = SHELL_4NQ
                      : dof-mapping : gdof[0] =  1 : gdof[1] =  2 : gdof[2] =  3
                                      gdof[3] =  4 : gdof[4] =  5 : gdof[5] =  6
                      : Young's Modulus =  E =        3.625e+03 ksi
                      : Yielding Stress = fy =        3.400e+03 psi
                      : Poisson's ratio = nu =        3.000e-01
                      : Density         =       0.000e+00 (null)
                      : Inertia Izz     =       0.000e+00 (null)
                      : Area            =       0.000e+00 (null)

EXTERNAL NODAL LOADINGS
-------------------------------------------------------------------------------
```

```
Node#    Fx (lbf)    Fy (lbf)  Fz (lbf)    Mx (lbf.in)  My (lbf.in)  Mz (lbf.in)
--------------------------------------------------------------------------------
   13    0.000       0.000     -455.729    0.000        0.000        0.000
  393    0.000       0.000     -455.729    0.000        0.000        0.000
  399    0.000       0.000     -455.729    0.000        0.000        0.000
   19    0.000       0.000     -455.729    0.000        0.000        0.000
   32    0.000       0.000     -911.458    0.000        0.000        0.000
   38    0.000       0.000     -911.458    0.000        0.000        0.000

   ..... details of nodal loads removed ....

  344    0.000       0.000    -1700.000    0.000        0.000        0.000
  350    0.000       0.000    -1700.000    0.000        0.000        0.000
  363    0.000       0.000    -1700.000    0.000        0.000        0.000
  369    0.000       0.000    -1700.000    0.000        0.000        0.000

============= End of Finite Element Mesh Description ==============

*** COMPUTE AND PRINT STIFFNESS AND EXTERNAL LOAD MATRICES

*** STATIC ANALYSIS PROBLEM

-------------------------------------------------------------------------------
 Node                                              Displacement
  No             displ-x           displ-y           displ-z              rot-x
-------------------------------------------------------------------------------
 units                in                in                in                rad
   1         1.33978e-01      -1.89314e-05      -1.80219e-03        2.02410e-04
   2         1.34131e-01      -2.29040e-05      -7.12851e-04        1.58344e-04
   3         1.34036e-01      -2.64397e-05       1.11846e-04        1.17222e-04
   4         4.80519e-04       7.93065e-05      -3.01849e-06        7.53268e-07
   5         0.00000e+00       0.00000e+00       0.00000e+00        0.00000e+00
  13         1.32863e-01      -3.34551e-05      -1.44068e-02        3.23153e-04

  ....... details of nodal displacements removed ......

 395         7.72668e-02      -5.87620e-05       1.43256e-03        3.45080e-05
 396         7.74839e-02      -9.59758e-13       1.76084e-03        5.23559e-16
 397         7.72668e-02       5.87620e-05       1.43256e-03       -3.45080e-05
 398         7.74887e-02      -1.71932e-05      -6.61126e-03       -2.90572e-04
 399         7.79492e-02       3.34551e-05      -1.44068e-02       -3.23153e-04
```

Figures 6.15 and 6.16 are contour and three-dimensional views of the bridge deck deflections, respectively. (the vertical axis of the deflection of Figure 6.16 has the units of inches). The bridge deflections are due to dead loads alone, and since the bridge geometry and section properties are symmetric, we expect that the deflections will also exhibit symmetry. They do.

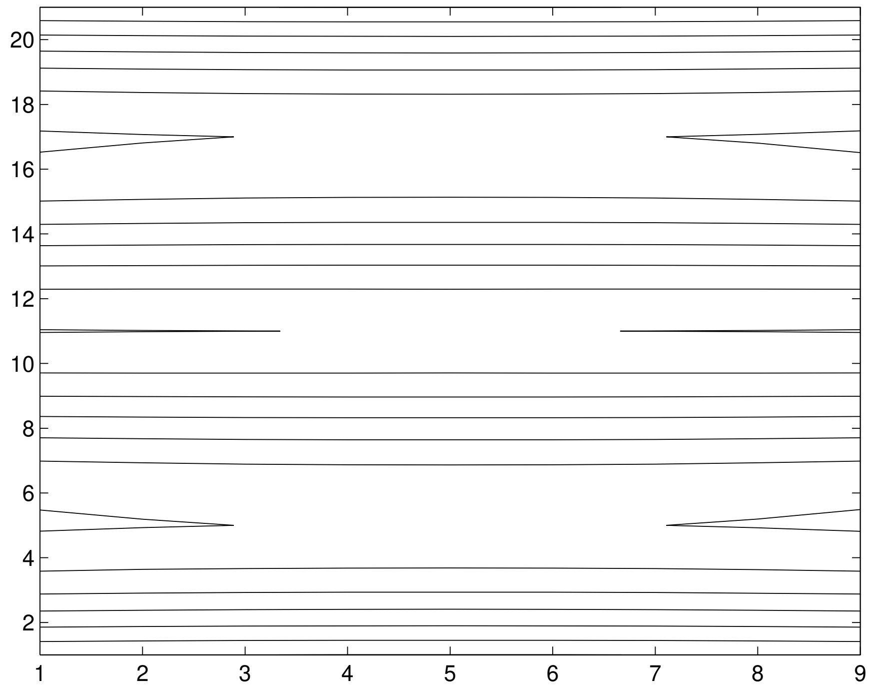


Figure 6.15: Highway Bridge : Contour Plot of Bridge Deck Deflections

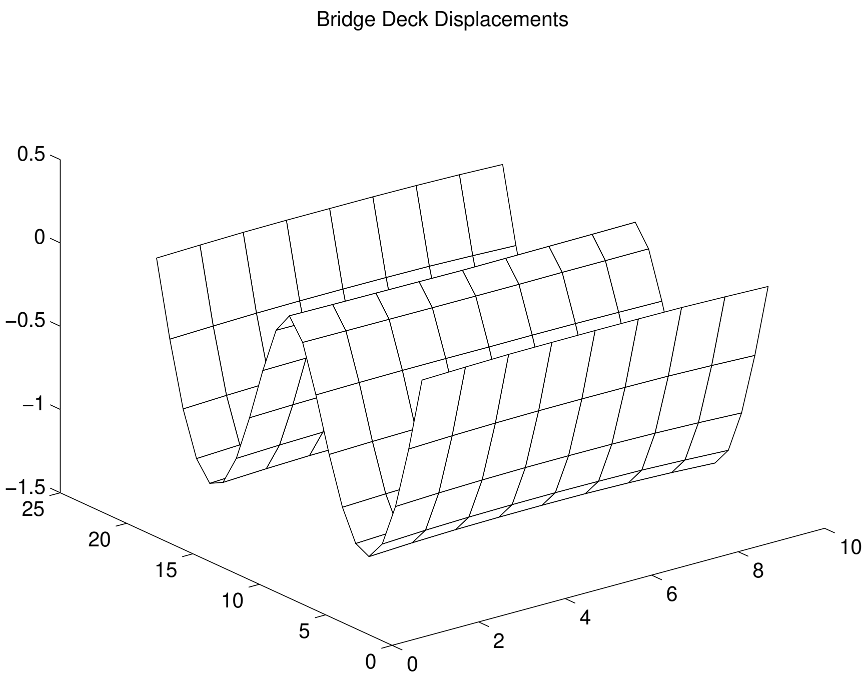


Figure 6.16: Three-Dimensional Mesh of Bridge Deck Deflections

**Moving Point Load Input File :** In Part 2 of our analysis, the latter sections of the input file are extended so that response evelopes are computed for a point load moving along the bridge.

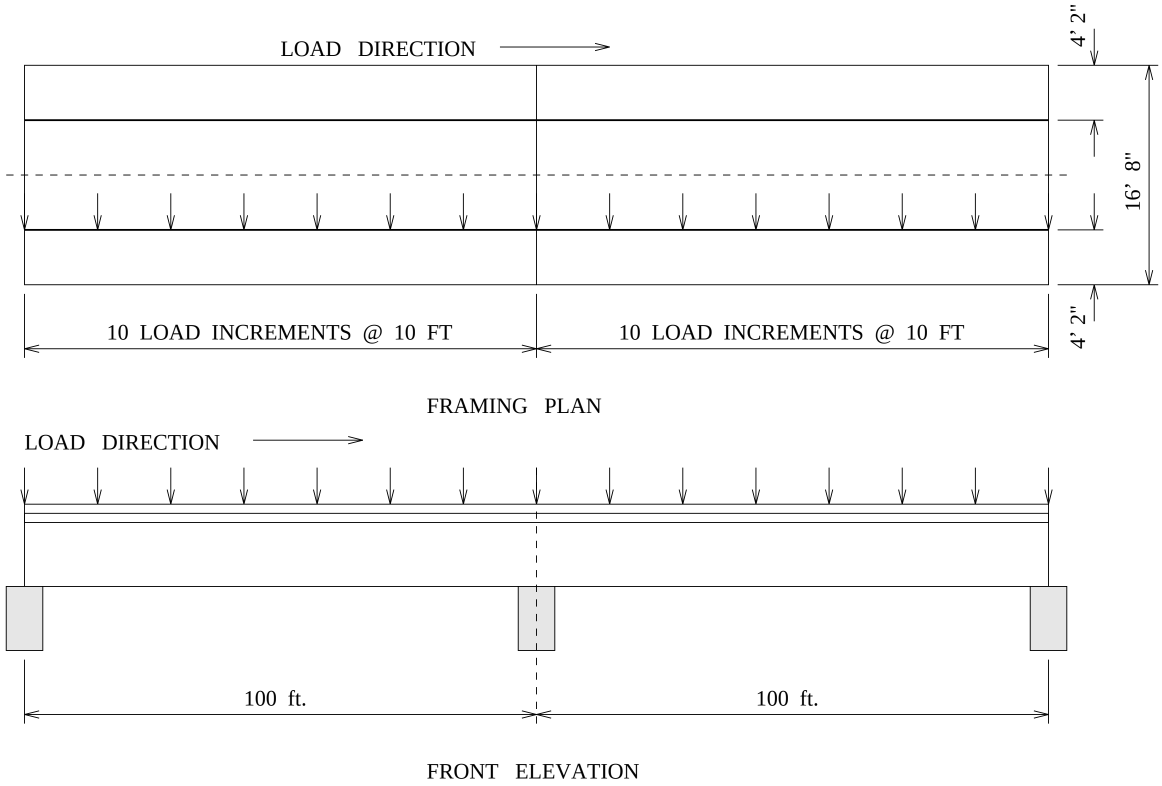


Figure 6.17: Plan and Front Elevation of Two-Span Continuous Highway Bridge with Moving Live Load

Figure 6.17 shows plan and elevation views of the bridge and moving point load. A cross sectional view of the bridge and the moving point load is shown in Figure 6.18. Abbreviated details of the input file are as follows:

START OF INPUT FILE

```
/*
 *  ===================================================================
 *  Analysis of Two Span Continuous Composite Beam (100-100ft), analysis
 *  assuming neutral axis of concrete deck coincide with top girder flange
 *  Moving load along one girder
 *
 *  Written By: Wane-Jang Lin                              December, 1994
 *  ===================================================================
 */

/* [a] : Setup problem specific parameters */
```

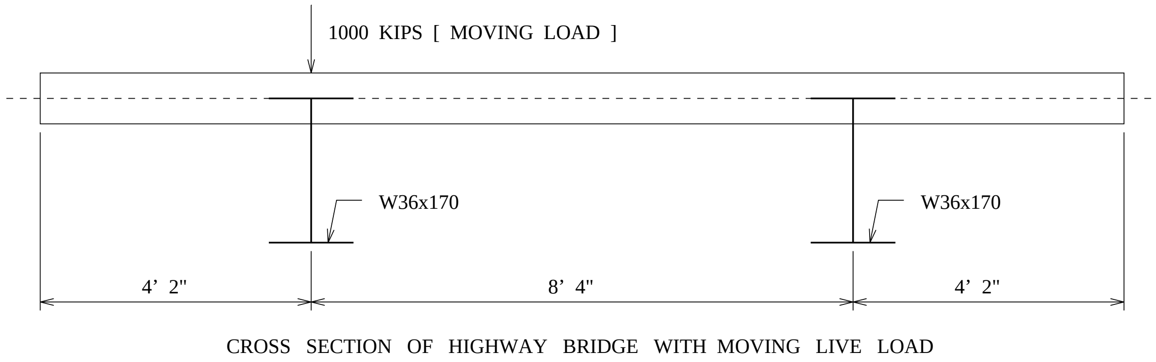


Figure 6.18: Cross Section of Two-Span Continuous Highway Bridge with Moving Live Load

```
   NDimension         = 3;
   NDofPerNode        = 6;
   MaxNodesPerElement = 4;

   StartMesh();

   print "*** GENERATE GRID OF NODES FOR FE MODEL \n\n";

         ..... details are the same as in static analysis datafile ....

   print "*** ATTACH ELEMENTS TO GRID OF NODES \n\n";

         ..... details are the same as in static analysis datafile ....

   print "*** DEFINE ELEMENT, SECTION AND MATERIAL PROPERTIES \n\n";

         ..... details are the same as in static analysis datafile ....

   print "*** SET UP BOUNDARY CONDITIONS \n\n";

         ..... details are the same as in static analysis datafile ....

/* [b] : Compile and Print Finite Element Mesh */

   EndMesh();
   PrintMesh();

/* [c] : Compute Stiffness Matrix */

   SetUnitsType("US");
   stiff = Stiff();

/* [d] : Compute Influence Lines for Moving Live Loads */

   print "\n*** STATIC ANALYSIS PROBLEM \n\n";

   Fx = 0 lbf;    Fy = 0 lbf;    Fz = -1000.0 kips;
```

```
Mx = 0 lbf*in; My = 0 lbf*in; Mz = 0 lbf*in;

nodeno1 = 97;
nodeno2 = 103;
step   = div_L*2+1;
print "node no 1 =",nodeno1,"\n";
print "node no 2 =",nodeno2,"\n";
print "Fz        =",Fz,"\n";
print "step      =",step,"\n";

influ_line1 = Zero([ step , 1 ]);  /* array for influence line 1 */
influ_line2 = Zero([ step , 1 ]);  /* array for influence line 2 */

load_node = 2;
lu = Decompose (stiff);
for( i=1 ; i<=step ; i=i+1 ) {
     NodeLoad( load_node, [Fx,Fy,Fz,Mx,My,Mz] );

     eload = ExternalLoad();
     displ = LUDecomposition( stiff, eload );

     node_displ_1 = GetDispl( [nodeno1], displ );
     node_displ_2 = GetDispl( [nodeno2], displ );

     influ_line1[i][1] = node_displ_1[1][3];
     influ_line2[i][1] = node_displ_2[1][3];

     NodeLoad( load_node, [-Fx,-Fy,-Fz,-Mx,-My,-Mz] );
     load_node = load_node + nodes_per_section;
}

PrintMatrix(influ_line1);
PrintMatrix(influ_line2);

quit;
```

Figure 6.19 shows the influence line of vertical displacement in the middle of one span (i.e. at node no 97) for the first girder subjected to 1000 kips moving live load. Similarly, Figure 6.20 shows the influence line of displacement in the middle of one span (i.e. node no 2 = 103) due to the 1000 kips moving live load.

The moving load analysis is one situation where a family of linear equations is solved with multiple right-hand sides. With this in mind, notice how we have called the function `Decompose()` once to decompose `stiff` into a product of upper and lower triangular matrices, and then called `LUDecomposition()` to compute the forward and backward substitution for each analysis. This strategy of equation solving reduces the overall solution time by approximately 70%.

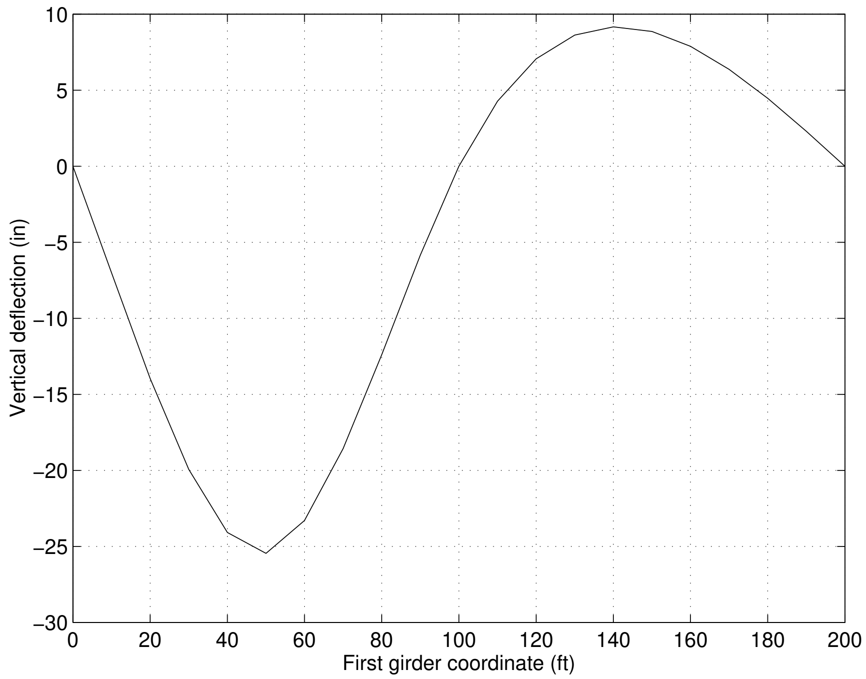


Figure 6.19: Influence line of displacement in the middle of one span for the first girder subjected to 1000 kips moving live load

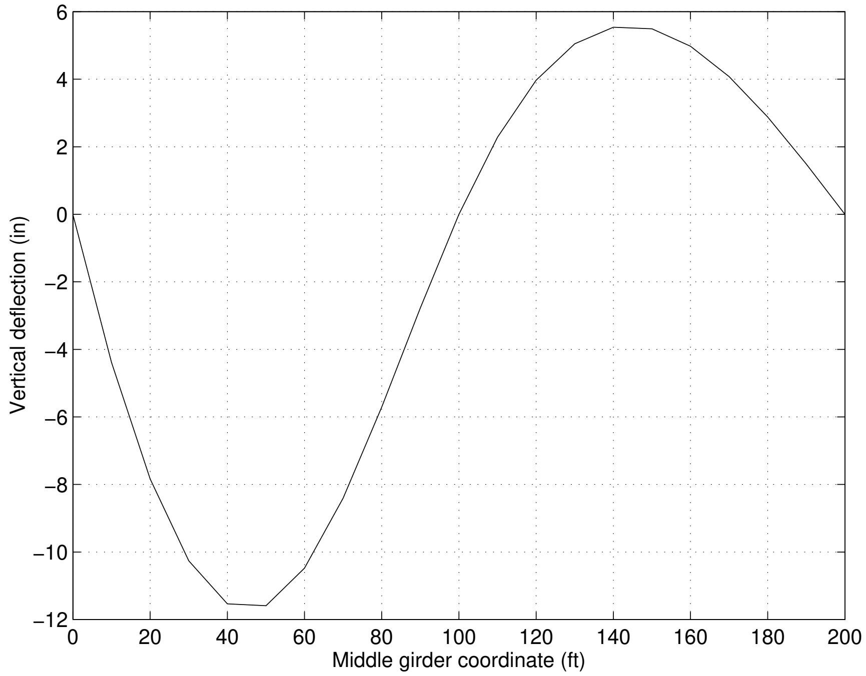


Figure 6.20: Influence line of displacement in the middle of one span for the second girder

# 6.2 Time-History Analyses

## 6.2.1 Modal Analysis of Five Story Steel Frame

**Description of Problems :** In this section we compute the linear time-history response of the steel frame structure described in Section 6.1.1. to dead and live gravity loads, plus a 1979 El Centro Earthquake ground motion scaled to moderate intensity.

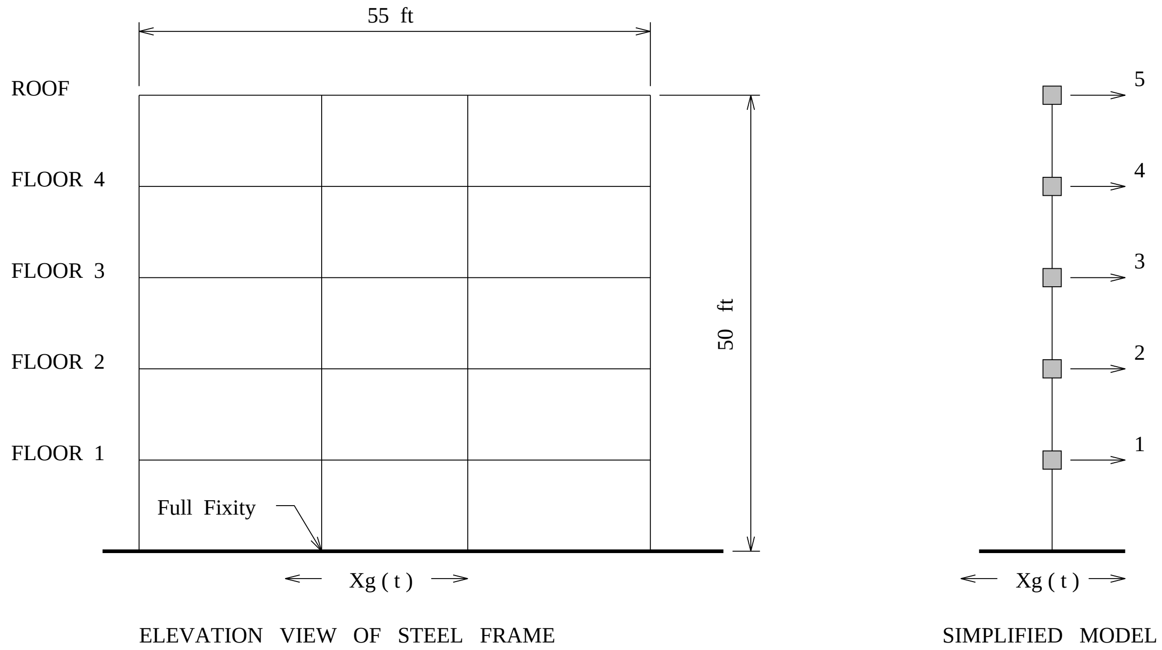


Figure 6.21: Elevation View and Simplified Model of Moment Resistant Frame subject to Scaled 1979 El Centro Earthquake

The left-hand side of Figure 6.21 is a simplified elevation view of the frame also shown in Figure 6.1. Unlike the analysis conducted in Section 6.1.1., we will assume that all nodal rotations and vertical deflections of the frame are fully restrained, and that the horizontal deflections across each floor may be lumped into a single degree of freedom. Together these assumptions imply that each floor will move as a rigid body in the horizontal direction, and that the global frame behavior may be described with only five degrees of freedom. The right-hand side of Figure 6.21 shows the simplified model that will be used in the earthquake time-history analysis – the shaded filled boxes represent the lumped masses at each floor level, and the global degrees of freedom begin at the first floor level and increase to the roof.

Figure 6.22 shows details of a 10 segment accelerograph extracted from the 1979 El Centro Earthquake ground motion. The peak ground acceleration is 86.63 cm/sec/sec. The ground motion will be scaled so that the peak ground acceleration is 0.15 g.

**Input File :** The following input file is based on the input file presented in Section 6.1.1,

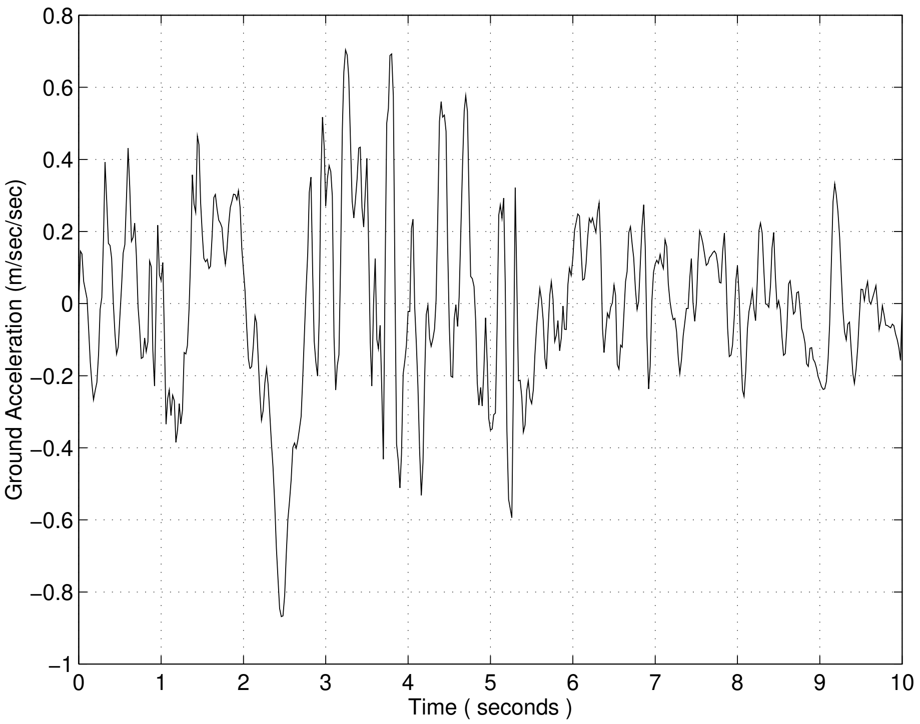


Figure 6.22: Ground Acceleration for 1979 El Centro Earthquake

with appropriate extensions for the earthquake analysis, and generation of time-history response quantities for post-analysis plotting.

START OF INPUT FILE

```
/*
 *  ============================================================
 *  Modal Analysis of Five Story Steel Moment Frame subject to
 *  El Centro ground motion.
 *
 *  Written By: Mark Austin                           July, 1995
 *  ============================================================
 */

/* [a] : Setup problem specific parameters */

   NDimension         = 2;
   NDofPerNode        = 3;
   MaxNodesPerElement = 2;

   StartMesh();

/* [b] : Generate two-dimensional grid of nodes */

   ..... same as for five story building ....

/* [c] : Attach column elements to nodes */
```

```
   ..... same as for five story building ....

/* [d] : Attach beam elements to nodes */

   ..... same as for five story building ....

/* [e] : Define section and material properties */

   ElementAttr("mrf_element") { type     = "FRAME_2D";
                                section  = "mrfsection";
                                material = "mrfmaterial";
                              }

   SectionAttr("mrfsection") { Izz       = 1541.9 in^4;
                               Iyy       =  486.3 in^4;
                               depth     =   12.0 in;
                               width     =   12.0 in;
                               area      =   47.4 in^2;
                             }

   MaterialAttr("mrfmaterial") { density = 7850 kg/m^3;
                                 poisson = 0.27;
                                 yield   = 275000 MPa;
                                 E       = 200000 MPa;
                               }

/* [f] : Apply full-fixity to columns at foundation level */

   for(nodeno = 1; nodeno <= 4; nodeno = nodeno + 1) {
       FixNode( nodeno, [ 1, 1, 1 ]);
   }

   for(nodeno = 5; nodeno <= 24; nodeno = nodeno + 1) {
       FixNode( nodeno, [ 0, 1, 1 ]);
   }

   LinkNode([  5,  6,  7,  8 ], [ 1, 0, 0] );
   LinkNode([  9, 10, 11, 12 ], [ 1, 0, 0] );
   LinkNode([ 13, 14, 15, 16 ], [ 1, 0, 0] );
   LinkNode([ 17, 18, 19, 20 ], [ 1, 0, 0] );
   LinkNode([ 21, 22, 23, 24 ], [ 1, 0, 0] );

/* [g] : Compile and Print Finite Element Mesh */

   EndMesh();
   PrintMesh();

/* [h] : Compute "stiffness" matrix; manually assemble "mass" matrices */

   stiff  = Stiff();
   PrintMatrix(stiff);

   mass = Zero([5,5]);
```

```
   mass = ColumnUnits ( mass , [N/m] );
   mass = RowUnits    ( mass , [sec^2] );

/* [i] : Manually assemble "mass" matrix : assume (dead + live) loads */

   dead_load       = 80 lbf/ft^2;
   floor_live_load = 40 lbf/ft^2;
   roof_live_load  = 20 lbf/ft^2;
   frame_spacing   = 20 ft;
   floor_length    = 55 ft;

   tributary_area = frame_spacing * floor_length;
   for(i = 1; i <= 5; i = i + 1) {
     if(i <= 4) then {
        floor_load = tributary_area * (dead_load + floor_live_load);
     } else {
        floor_load = tributary_area * (dead_load + roof_live_load);
     }

     mass [i][i] = mass[i][i] + floor_load / (32.2 ft/sec^2);
   }

   PrintMatrix(mass);

/* [j] : Compute and print eigenvalues and eigenvectors */

   no_eigen = 2;
   eigen       = Eigen(stiff, mass, [ no_eigen ]);
   eigenvalue  = Eigenvalue(eigen);
   eigenvector = Eigenvector(eigen);

   for(i = 1; i <= no_eigen; i = i + 1) {
       print "Mode", i ," : w^2 = ", eigenvalue[i][1];
       print " : T = ", 2*PI/sqrt(eigenvalue[i][1]) ,"\n";
   }

   PrintMatrix(eigenvector);

/* [k] : Generalized mass and stiffness matrices */

   EigenTrans = Trans(eigenvector);
   Mstar   = EigenTrans*mass*eigenvector;
   Kstar   = EigenTrans*stiff*eigenvector;

   PrintMatrix( Mstar );
   PrintMatrix( Kstar );

/* [m] : Setup Rayleigh Damping for Frame Structure */

   rdamping = 0.05;
   W1 = sqrt ( eigenvalue[1][1]);
   W2 = sqrt ( eigenvalue[2][1]);

   A0 = 2*rdamping*W1*W2/(W1 + W2);
```

```
   A1 = 2*rdamping/(W1 + W2);

   print "A0 = ", A0, " A1 = ", A1, "\n";

   Cstar = A0*Mstar + A1*Kstar;
   PrintMatrix( Cstar );

/* [n] : Define earthquake loadings... */

   Elcentro = ColumnUnits( [
     14.56;    13.77;     6.13;     3.73;     1.32;    -6.81;   -16.22;   -22.41;

     ...... details of earthquake removed .....

    -10.05;   -12.35;   -15.72;     0.00  ], [cm/sec/sec] );

   ground_motion_scale_factor = 0.15*981.0/86.63;

   PrintMatrix( Elcentro );

/* [o] : Initialize system displacement, velocity, and load vectors */

   displ  = ColumnUnits( Matrix([5,1]), [m]    );
   vel    = ColumnUnits( Matrix([5,1]), [m/sec]);
   eload  = ColumnUnits( Matrix([5,1]), [kN]);
   r      = One([5,1]);

/* [p] : Initialize modal displacement, velocity, and acc'n vectors */

   Mdispl  = ColumnUnits( Matrix([ no_eigen,1 ]), [m]    );
   Mvel    = ColumnUnits( Matrix([ no_eigen,1 ]), [m/sec]);
   Maccel  = ColumnUnits( Matrix([ no_eigen,1 ]), [m/sec/sec]);

/*
 * [q] : Allocate Matrix to store five response parameters --
 *       Col 1 = time (sec);
 *       Col 2 = 1st mode displacement (cm);
 *       Col 3 = 2nd mode displacement (cm);
 *       Col 4 = 1st + 2nd mode displacement (cm);
 *       Col 5 = Total energy (Joules)
 */

   dt     = 0.02 sec;
   nsteps = 600;
   beta   = 0.25;
   gamma  = 0.5;

   response = ColumnUnits( Matrix([nsteps+1,5]), [sec], [1]);
   response = ColumnUnits( response,  [cm], [2]);
   response = ColumnUnits( response,  [cm], [3]);
   response = ColumnUnits( response,  [cm], [4]);
   response = ColumnUnits( response, [Jou], [5]);

/* [r] : Compute (and compute LU decomposition) effective mass */
```

```
   MASS  = Mstar + rdamping*dt*Cstar + Kstar*beta*dt*dt;
   lu    = Decompose(Copy(MASS));

/* [s] : Mode-Displacement Solution for Response of Undamped MDOF System  */

   MassTemp = -mass*r;
   for(i = 1; i <= nsteps; i = i + 1) {
       print "*** Start Step ",i,"\n";

       if(i == 2) {
          SetUnitsOff;
       }

    /* [s.1] : Update external load */

       if(i <= 500) then {
          eload = MassTemp*Elcentro[i][1]*ground_motion_scale_factor;
       } else {
          eload = MassTemp*(0)*ground_motion_scale_factor;
       }

       Pstar = EigenTrans*eload;

       R = Pstar - Kstar*(Mdispl + Mvel*dt + Maccel*(dt*dt/2.0)*(1-2*beta)) -
           Cstar*(Mvel + Maccel*dt*(1-gamma));

    /* [s.2] : Compute new acceleration, velocity and displacement  */

       Maccel_new = Substitution(lu,R);
       Mvel_new   = Mvel   + dt*(Maccel*(1.0-gamma) + gamma*Maccel_new);
       Mdispl_new = Mdispl + dt*Mvel + ((1 - 2*beta)*Maccel +
                    2*beta*Maccel_new)*dt*dt/2;

    /* [s.3] : Update and print new response */

       Maccel = Maccel_new;
       Mvel   = Mvel_new;
       Mdispl = Mdispl_new;

    /* [s.4] : Combine Modes */

       displ = eigenvector*Mdispl;
       vel   = eigenvector*Mvel;

    /* [s.5] : Compute Total System Energy */

       a1 = Trans(vel);
       a2 = Trans(displ);
       e1 = a1*mass*vel;
       e2 = a2*stiff*displ;
       energy = 0.5*(e1 + e2);

    /* [s.6] : Save components of time-history response */
```

```
      response[i+1][1] = i*dt;                              /* Time                  */
      response[i+1][2] = eigenvector[1][1]*Mdispl[1][1];  /* 1st mode displacement */
      response[i+1][3] = eigenvector[1][2]*Mdispl[2][1];  /* 2nd mode displacement */
      response[i+1][4] = displ[1][1];               /* 1st + 2nd mode displacement */
      response[i+1][5] = energy[1][1];                      /* System Energy         */
   }

/* [t] : Print response matrix and quit */

   SetUnitsOn;
   PrintMatrix(response);
   quit;
```

Points to note are:

[1] Notice that we have used the finite element routines to compute the stiffness matrix, but have manually assembled the mass matrix. It is assumed that when the earthquake occurs, the frame loads will equal the dead loading plus the full live loading. The tributary area equals the frame spacing (20 ft) times the frame width (55 ft).

[2] The global mass and stiffness matrices are $(5 \times 5)$ matrices. The time-history analysis is simplified by computing the Newmark integration on the first and second modes of the frame alone. The generalized mass and stiffness matrices are as described in Section 4.3.

[3] The damping matrix is taken as a linear combination of the generalized mass and stiffness matrices, namely:

$$\mathbf{C}^* = a_o \cdot \mathbf{M}^* + a_1 \cdot \mathbf{K}^*. \tag{6.3}$$

Let $w_m$ and $\xi_m$ be the circular natural frequency and ratio of critical damping in the mode $m$, and $w_n$ and $\xi_n$ be the circular natural frequency and ratio of critical damping in the mode $n$. It can be shown that the coefficients $a_1$ and $a_2$ are given by:

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \frac{2 \cdot w_m \cdot w_n}{w_n^2 - w_m^2} \cdot \begin{bmatrix} w_n & -w_m \\ -1/w_n & 1/w_m \end{bmatrix} \cdot \begin{bmatrix} \xi_m \\ \xi_n \end{bmatrix} \tag{6.4}$$

For this analysis we will let $m = 1$ and $n = 2$, and assume that the critical ratio of damping is the same in both modes. The input file variable `rdamping` represents the ratio of critical damping in the first and second modes. Analyses will be conducted for `rdamping = 0.0`, and `rdamping = 0.05`.

[4] The time-history analysis is based on the Newmark equations given in Section 4.2, and the modal analysis equations specified in Section 4.3. At each step the analysis we compute solutions to the equation of equilibrium

$$\mathbf{\Phi^T M \Phi} \ddot{Y}(t) + \mathbf{\Phi^T C \Phi} \dot{Y}(t) + \mathbf{\Phi^T K \Phi} Y(t) = \mathbf{\Phi^T} P(t) = -\mathbf{\Phi^T M r} \ddot{X}_g(t). \quad (6.5)$$

Here $\mathbf{\Phi}$, $\mathbf{M}$, $\mathbf{C}$, and $\mathbf{K}$ are as previously defined. $\mathbf{r}$ is a $(5 \times 1)$ that describes the movement in each of the frame degrees of freedom due to a unit ground displacement – in this case, $\mathbf{r} = (5 \times 1)$ vector of ones. $\ddot{X}_g(\mathrm{t})$ is the ground acceleration at time $t$.

[5] The response is computed for 600 steps of time-step 0.02 seconds (i.e. 12 seconds). The first loop of analysis is conducted with the units checking turned on – thereafter, the units are switched off. Constant acceleration across each time-step is achieved by setting the Newmark parameters $\beta = 0.25$ and $\gamma = 0.5$ – see Section 4.2 for a detailed discussion on the Newmark method.

[6] At the end of each time-step we compute the sum of kinetic plus potential energy for the frame's first two modes. Theoretical considerations indicate that when $\beta = 0.25$ and $\gamma = 0.5$, the total energy of the undamped system will be conserved after the earthquake motion has stopped (i.e. the last 2 seconds of the computed response).

[7] We have allocated a 600 by 5 response matrix to stored selected components of the time-history response. See the input file for a description of the contents in each column.

**Abbreviated Output File :** The following output file is a summary of the output generated for the simulation case `rdamping` = 0.0.

START OF ABBREVIATED OUTPUT FILE

```
===========================================
Title : DESCRIPTION OF FINITE ELEMENT MESH
===========================================

=======================
Profile of Problem Size
=======================

   Dimension of Problem        =      2

   Number Nodes                =     24
   Degrees of Freedom per node =      3
   Max No Nodes Per Element    =      2

   Number Elements             =     35
   Number Element Attributes   =      1
   Number Loaded Nodes         =      0
   Number Loaded Elements      =      0

------------------------------------------------------------
Node#      X_coord           Y_coord          Tx     Ty     Rz
------------------------------------------------------------
```

```
    1    0.0000e+00 ft    0.0000e+00 ft    -1    -2    -3
    2    2.0000e+01 ft    0.0000e+00 ft    -4    -5    -6
    3    3.5000e+01 ft    0.0000e+00 ft    -7    -8    -9
    4    5.5000e+01 ft    0.0000e+00 ft   -10   -11   -12

    ..... details of nodal coordinates removed .....

   24    5.5000e+01 ft    5.0000e+01 ft     5   -51   -52

-------------------------------------------------------------------
Element#     Type         node[1]   node[2]      Element_Attr_Name
-------------------------------------------------------------------

       1   FRAME_2D             1         5      mrf_element
       2   FRAME_2D             5         9      mrf_element

       ..... details of frame elements removed .....

      34   FRAME_2D            22        23      mrf_element
      35   FRAME_2D            23        24      mrf_element

------------------------
Element Attribute Data :
------------------------

ELEMENT_ATTR No.   1  : name = "mrf_element"
                      : section = "mrfsection"
                      : material = "mrfmaterial"
                      : type = FRAME_2D
                      : gdof [0] =    1 : gdof[1] =    2 : gdof[2] =    3
                      : Young's Modulus =  E =        2.000e+05 MPa
                      : Yielding Stress = fy =        2.750e+05 MPa
                      : Poisson's ratio = nu =        2.700e-01
                      : Density         =        7.850e+03 kg/m^3
                      : Inertia Izz     =        1.542e+03 in^4
                      : Area            =        4.740e+01 in^2

============= End of Finite Element Mesh Description ==============

SKYLINE MATRIX : "stiff"

row/col                 1             2             3             4             5
       units           N/m           N/m           N/m           N/m           N/m
   1          4.35158e+08  -2.17579e+08   0.00000e+00   0.00000e+00   0.00000e+00
   2         -2.17579e+08   4.35158e+08  -2.17579e+08   0.00000e+00   0.00000e+00
   3          0.00000e+00  -2.17579e+08   4.35158e+08  -2.17579e+08   0.00000e+00
   4          0.00000e+00   0.00000e+00  -2.17579e+08   4.35158e+08  -2.17579e+08
   5          0.00000e+00   0.00000e+00   0.00000e+00  -2.17579e+08   2.17579e+08

MATRIX : "mass"

row/col                 1            2            3            4            5
       units           N/m          N/m          N/m          N/m          N/m
```

```
   1    sec^2   5.98259e+04  0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00
   2    sec^2   0.00000e+00  5.98259e+04  0.00000e+00  0.00000e+00  0.00000e+00
   3    sec^2   0.00000e+00  0.00000e+00  5.98259e+04  0.00000e+00  0.00000e+00
   4    sec^2   0.00000e+00  0.00000e+00  0.00000e+00  5.98259e+04  0.00000e+00
   5    sec^2   0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00  4.98550e+04

*** SUBSPACE ITERATION CONVERGED IN  9 ITERATIONS
Mode         1  : w^2 =        313 1/sec^2  : T =     0.3551 sec
Mode         2  : w^2 =       2648 1/sec^2  : T =     0.1221 sec

MATRIX : "eigenvector"

row/col                  1            2
        units
   1            2.91629e-01  7.87127e-01
   2            5.58155e-01  1.00000e+00
   3            7.76637e-01  4.84597e-01
   4            9.28270e-01 -3.82551e-01
   5            1.00000e+00 -9.70739e-01

MATRIX : "Mstar"

row/col                  1            2
        units          N/m          N/m
   1    sec^2   1.61217e+05  7.27596e-12
   2    sec^2   7.27596e-12  1.66677e+05

MATRIX : "Kstar"

row/col                  1            2
        units          N/m          N/m
   1            5.04687e+07  2.04891e-08
   2            0.00000e+00  4.41345e+08
A0 =           0 1/sec  A1 =           0 sec

MATRIX : "Cstar"

row/col                  1            2
        units     kg/sec^3     kg/sec^3
   1    sec^2   0.00000e+00  0.00000e+00
   2    sec^2   0.00000e+00  0.00000e+00

MATRIX : "response"

row/col                  1            2            3            4            5
        units          sec           cm           cm           cm          Jou
   1            0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00  0.00000e+00
   2            2.00000e-02 -5.17642e-04 -3.51330e-04 -8.68972e-04  2.82918e-01

   ...... details of response matrix removed : for details, see the figures.

 600            1.19800e+01 -4.21771e-02 -2.57188e-02 -6.78958e-02  3.88629e+02
 601            1.20000e+01 -7.47478e-02 -1.97584e-02 -9.45062e-02  3.88629e+02
```

Matrices `Mstar` and `Kstar` in the program output correspond to the generalized mass and stiffness matrices, respectively. We expect that the computed eigenvectors will be orthogonal with respect to the mass and stiffness matrices, and hence, the generalized mass and stiffness matrices will be (nearly) diagonal – we see that in practice, some round-off errors occur.

The first and second modes of vibration have natural periods `0.3551` sec and `0.1221` sec, respectively. When the frame is undamped, coefficients $a_o$ and $a_1$ evaluate to zero, as expected.

Figures 6.23 to 6.26 summarize the time-history response of the undamped moment resistant frame. Figure 6.23 is the time history response for mode 1, Figure 6.24 the time history response for mode 2, and Figure 6.25 the time history response for modes 1 and 2 combined. Figure 6.26 plots the kinetic plus potential energy of the frame – you should notice that once the ground motion has ceased (at t = 10 seconds), the Newmark method conserves energy.

Figures 6.27 and 6.27 plot the total time-history response and energy when the frame has 5% damping. After the ground motion has ceased, the roof displacement decreases steadily to zero. You can easily observe that between t = 10 seconds and t = 12 seconds, the roof oscillates through approximately 5.5 cycles. This implies an approximate natural frequency of $2/5.5 \approx 0.36$ seconds.

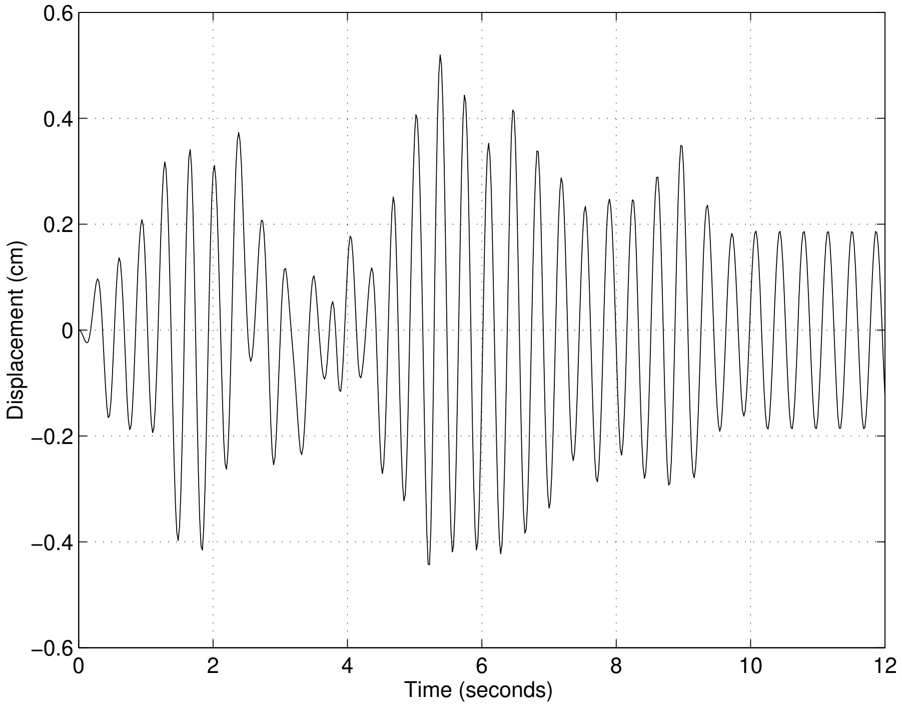


Figure 6.23: Modal Analysis : First Mode Displacement of Roof (cm) versus Time (sec)

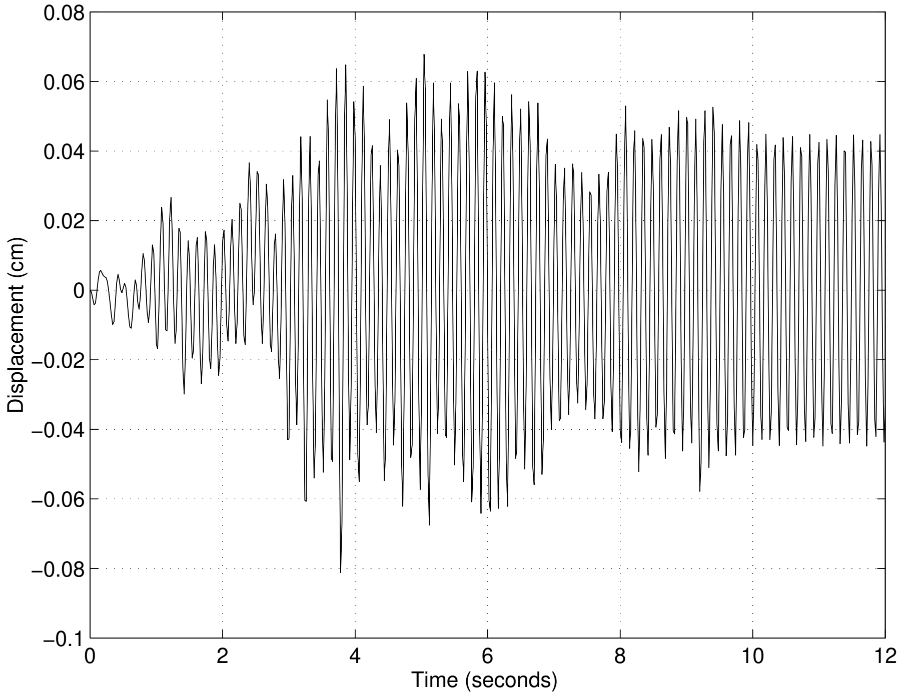


Figure 6.24: Modal Analysis : Second Mode Displacement of Roof (cm) versus Time (sec)

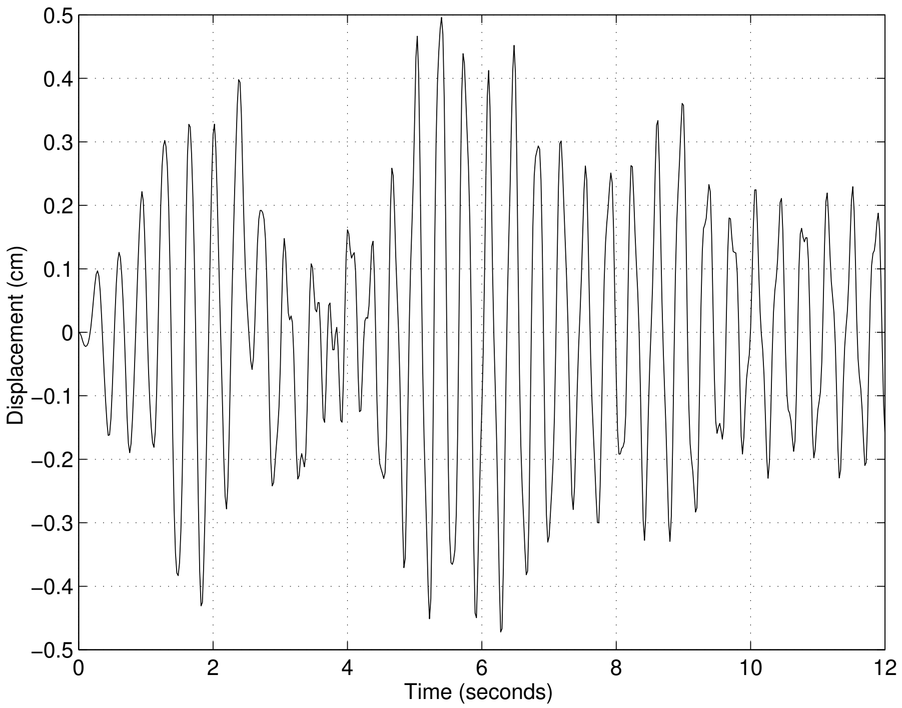


Figure 6.25: Modal Analysis : First + Second Mode Displacement of Roof (cm) versus Time (sec)

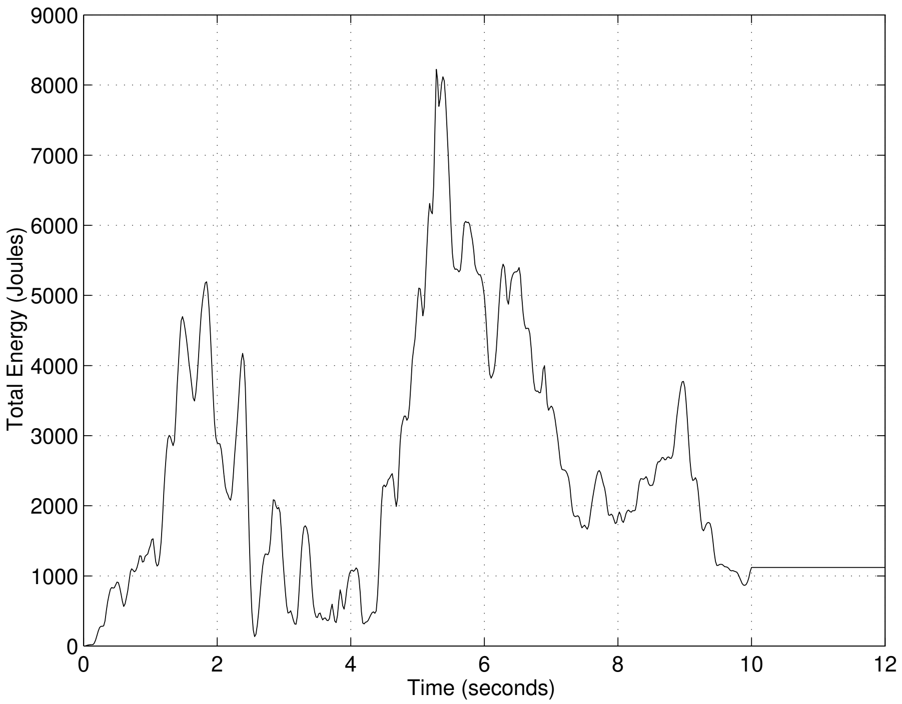


Figure 6.26: Modal Analysis : Total Energy (Joules) versus Time (sec)

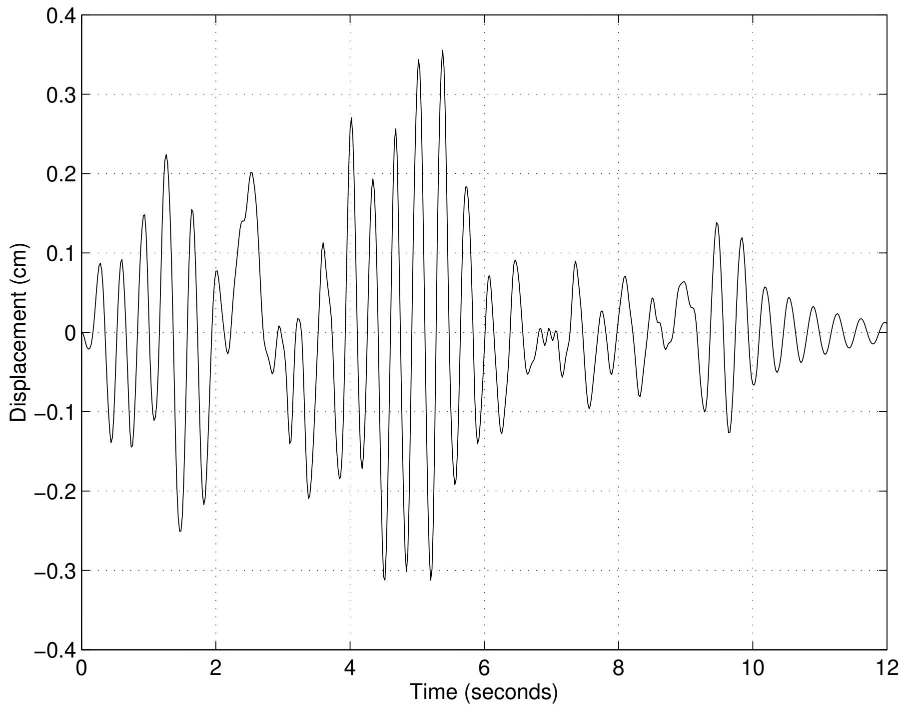


Figure 6.27: Modal Analysis at 5% damping : First + Second Mode Displacement of Roof (cm) versus Time (sec)

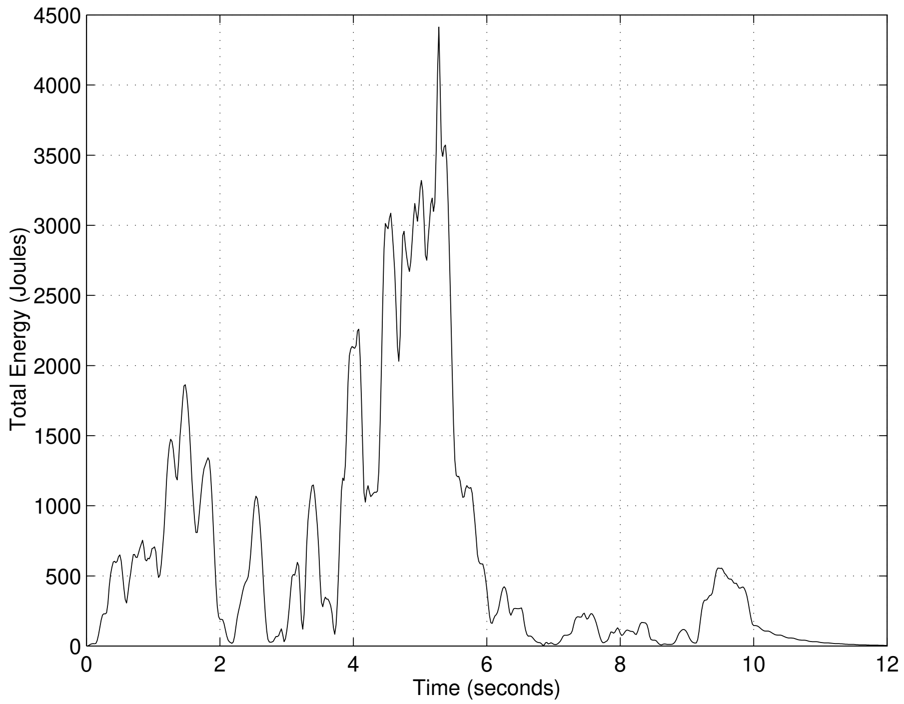


Figure 6.28: Modal Analysis at 5% damping : Total Energy (Joules) versus Time (sec)

# Part IV

# ARCHITECTURE AND DESIGN

# Chapter 7

# Data Types : Physical Quantity and Matrix Data Structures

## 7.1 Introduction

Basic engineering quantities such as, length, mass, and force, are defined by a numerical value (number itself) plus physical units. The importance of the engineering units is well known. Unfortunately, physical units are not a standard part of main-stream finite element software packages – indeed, most engineering software packages simply assume that engineering units will be consistent, and leave the details of checking to engineer (in the writers' experience, it is not uncommon for engineers to overlook this detail). While this practice of implementation may be satisfactory for computation of well established algorithms, it is almost certain to lead to incorrect results when engineers are working on the development of new and innovative computations.

ALADDIN deviates from this trend by supporting the basic data type physical quantity, as well as matrices of physical quantities. Engineers may define units as part of the problem description, and manipulate these quantities via standard arithmetic and matrix operations. Any set of valid engineering units can be used. For example, *mm*, *cm*, *m*, *km*, and *in*, are all valid units of length. In this chapter, we discuss the data structures used to implement physical quantities, and matrices of physical quantities. Special storage techniques for large symmetric matrices are also mentioned.

## 7.2 Physical Quantities

Any physical unit can be decomposed into basic units. The four basic units needed for engineering analysis are: length unit $\mathcal{L}$; mass unit $\mathcal{M}$; time unit t; and temperature unit $\mathcal{T}$. Any engineering unit can be obtained with the following combinations:

$$unit = k\mathcal{L}^{\alpha}\mathcal{M}^{\beta}t^{\gamma}\mathcal{T}^{\delta}.$$

where $\alpha, \beta, \gamma, \delta$ are exponents, and k is the scale factor. For units of length and mass, $[\alpha, \beta, \gamma, \delta] = [1, 0, 0, 0]$ and $[\alpha, \beta, \gamma, \delta] = [0, 1, 0, 0]$, respectively. Non-dimensional quantities (i.e. numbers, degrees, and radians) are given by the family of zero exponents $[\alpha, \beta, \gamma, \delta] = [0, 0, 0, 0]$. With this basic units in place, we can define the units data structure as:

```
#define  SI     100
#define  US     200
#define  SI_US  300

typedef struct dimensional_exponents {
        char         *units_name; /* units name              */
        double      scale_factor; /* scale/conversion factor  */
        double      length_expnt; /* exponent for length      */
        double        mass_expnt; /* exponent for mass        */
        double        time_expnt; /* exponent for time        */
        double        temp_expnt; /* exponent for temperature */
        int           units_type; /* US or SI units           */
} DIMENSIONS;
```

The character string stores the unit name, and scale_factor, the scale factor with respect to a basic unit. For example, in US units the basic unit of length is inch. It follows that 12 will be the scale_factor for one foot. The units_type flag allows for the representation of various systems of units (e.g. SI, US, and combined SI_US). SI_US units are those shared by both US and SI systems – the two most important examples being time and non-dimensional units. We use variables of data type double to represent exponents, thereby avoiding mathematical difficulties in manipulation of quantities. For example, if $x = y^{0.5}$, the unit exponents of x, $[\alpha_x, \beta_x, \gamma_x, \delta_x] = 0.5 \times [\alpha_y, \beta_y, \gamma_y, \delta_y]$. If the units exponents were defined as integers, then x would be incorrectly truncated to $[\alpha_x, \beta_x, \gamma_x, \delta_x] = [0, 0, 0, 0]$. With the units data structure in place, the quantity data structure is simply defined as

```
typedef struct engineering_quantity {
        double       value;
        DIMENSIONS  *dimen;
} QUANTITY;
```

Notice that physical units alone are the special case of a quantity with numerical value 1.0.

### 7.2.1 Relationship between Quantity and Units

In engineering applications, there is no need to distinguish between different systems of units during calculation. The main use of units is for clarification of problem input and problem output. ALADDIN stores a physical quantity as an *unscaled value* with respect to a set of reference units, and by default, all quantities are stored internally in the SI units system. In the SI system, the reference set of units are meter "m" for length, kilogram "kg" for mass, second "sec" for time and "deg_C" for temperature.

Suppose, for example, that we define the quantity, x = 1 km (one kilometer). Because 1 km = 1000 m, x will be saved internally as value = 1000, In the data structure `DIMENSIONS`, units_name will point to the character string "km". The scale_factor for "km" with respect to reference unit "m" is 1000. Now lets see what happens when we print x. Because the output value = value/scale_factor = 1. "print x" will still give "1 km".

### 7.2.2 US and SI Units Conversion

Scale factors are needed to convert units in US to SI, and vice versa. All quantities in US units must be converted into SI before they can be used in calculations. Fortunately, for most sets of units, only a single scale factor is needed to convert a quantity from US into SI and vice versa (e.g. 1 in = 24.5 mm = 25.4E-3 m).

The important exception is temperature units. Conversion of temperature units is complicated by the nonlinear relationship between systems – that is $x^oF \neq x \times 1^oF$, since $x^oF = (5/9)(x-32) \times 1^oC$. In other words, the value and unit of a temperature quantity can not be treated separately. We have to know the value of temperature before we can convert it's units from one to another unit system. ALADDIN handles this problem by first installing the "deg_F" (unit for $^oF$) with scale_factor = 1 into symbol table. When the value (a number) and the unit are combined into a quantity we convert the deg_F into deg_C. When a temperature quantity is printed out, we need to convert it's units to the prefered units. A related complication is units for temperature increment. One degree of temperature increase is different from one degree temperature. To see how this problem arises, let $y = 1^oF$. If y is a one degree temperature increase, then $y = (\alpha+1)^oF - \alpha^oF = 5/9^oC$, where $\alpha$ is value of previous temperature. If, on the other hand, y is at one degree Fahrenheit temperature, then $y = (5/9)(1-32)^oC = -17.22^oC$. Similar difficulties crop up in the computation of temperature gradient – $10mm/^oF$ can only be considered as a 10 mm extension for one Fahrenheit degree temperature increase, which is $18mm/^oC$. This cannot be considered as $(10mm)/(1^oF)$, which is $(-0.5806mm)/(1^oC)$ (divisions of this kind in real engineering computations are meaningless). To avoid these confusions, a new set of units are chosen for temperature increments, they are "DEG_F and DEG_C" to distinguish from "deg_F and deg_C". The conversion between DEG_F and DEG_C is DEG_F = 5/9 DEG_C. And for simplicity, we will ignore the case of $(10mm)/(1^oF)$.

**Library Functions for Internal Operations on Units :** ALADDIN also provides functions for internal operations on units – example functions include copying units, obtaining default units, checking consistency of units, simplifying the units name after various operations, calculating the length of units name, and printing of units. For a summary of functions, see Table 7.1. `Note :` Some mathematical functions such as log(x), ln(x) or exp(x), cannot have units in their argument lists. Other mathematical functions, such as sine and cosine, can only have degree or radian in their arguments.

| Functions for Units Operations | |
|---|---|
| Function | Purpose of Function |
| SameUnits() | Check if two units are of same type |
| UnitsMult() | Multiply two units |
| UnitsDiv() | Divide two units |
| UnitsCopy() | Make a copy of a unit |
| UnitsPower() | compute a unit raised to a power |
| UnitsNegate() | Negate a unit d = 1/d |
| UnitsPrint() | Print name, scalefactor, type and exponents of a unit |
| ZeroUnits() | Initialize a unit |
| DefaultUnits() | Determine pointer on symbol table for a given unit name |
| UnitsSimplify() | Simplify unit name string with simple character string |
| RadUnitsSimplify() | Convert a nondimensional unit into a radian unit |
| UnitsLength() | Measure the total length of several units names |
| ConvertTempUnits() | Convert temperature units between SI and US systems |
| UnitsConvert() | Convert units between US and SI systems – except for temperature units |

Table 7.1: **Functions for Units Operations**

## 7.3 Matrices

ALADDIN's matrix module has following features:

[1] It supports wide range of matrix operations, data types (integer, double and complex), and storage methods. In this section, only the double and indirect storage methods are discussed. The skyline matrix for large degree of freedom will be discussed in next section.

[2] It allows users to dynamic allocate and deallocate the memory of matrix.

[3] It provides the units for each element of the matrix.

Our strategy of development is to devise a software framework that will allow any combination of data type and storages scheme to be implemented.

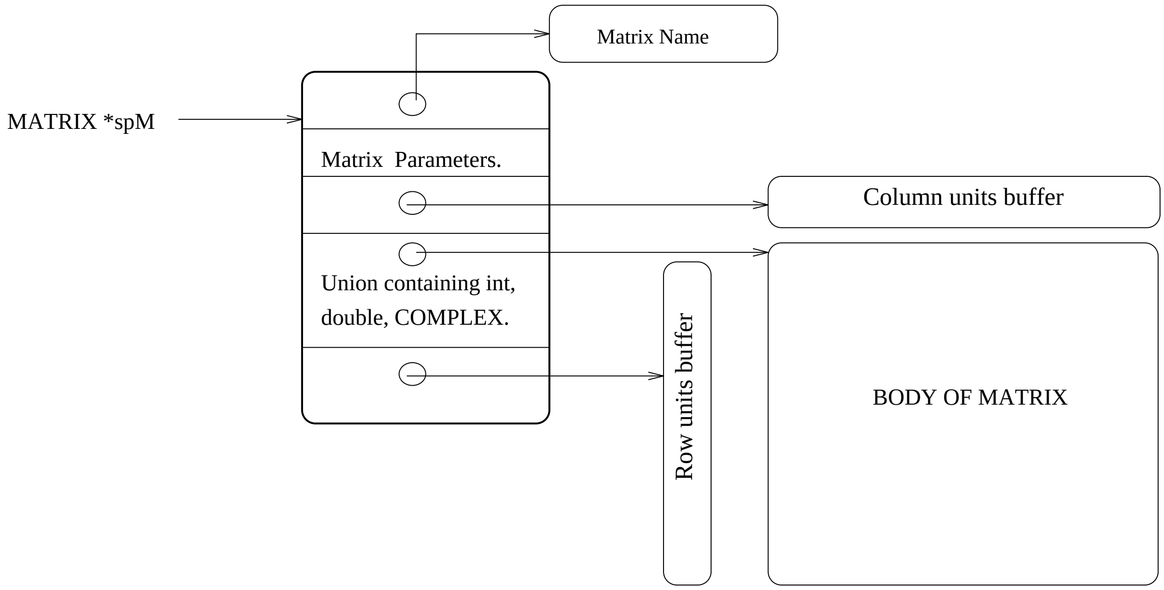


Figure 7.1: Layout of Memory in Matrix Data Structure

Figure 7.1 shows a high-level layout of memory for the matrix data structure. Details of the matrix data structure are as follows:

```
/* Data Structures for Matrices of Engineering Quantities */

typedef struct {
        double dReal, dImaginary;
} COMPLEX;

typedef enum {
```

```
        INTEGER_ARRAY  = 1,
        DOUBLE_ARRAY   = 2,
        COMPLEX_ARRAY  = 3
} DATA_TYPE;

typedef enum {
        SEQUENTIAL = 1,
        INDIRECT   = 2,
        SKYLINE    = 3,
        SPARSE     = 4
} INTERNAL_REP;

typedef struct MATRIX {
        char      *cpMatrixName;    /*  *name           */
        int             iNoRows;    /*  no_rows         */
        int          iNoColumns;    /*  no_columns      */
        DIMENSIONS  *spRowUnits;    /*  *row_units_buf  */
        DIMENSIONS  *spColUnits;    /*  *col_units_buf  */
        INTERNAL_REP       eRep;    /*  storage type    */
        DATA_TYPE         eType;    /*  data type       */
        union {
            int           **iaa;
            double        **daa;
            COMPLEX       **caa;
        } uMatrix;
} MATRIX;
```

Memory is provided for a character string to the matrix name, two integers for the number of matrix rows and columns, two one-dimensional arrays of data type `DIMENSIONS`, and integer flags for the basic data type and storage scheme for matrix elements. The union `uMatrix` contains pointers to matrix bodies of integers, doubles and `COMPLEX` elements. For the purpose of this study, however, we will implement and describe algorithms for data type double plus units alone (i.e. `DATA_TYPE` equals `DOUBLE_ARRAY`).

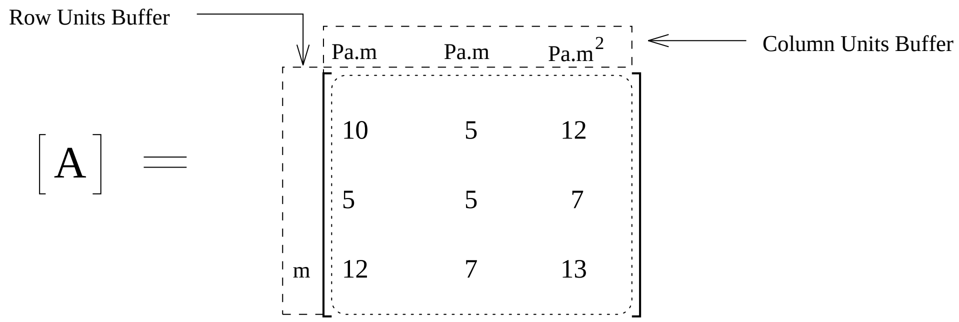


Figure 7.2: Matrix with Units Buffers

The units for elements in a matrix are stored in two one-dimensional arrays of data type DIMENSIONS. One array stores column units, and the second array row units. The units for matrix element at row **i** and column `j` is simply the product of the `i`$^{th}$ element of the row units buffer and the j$^{th}$ element of column units buffer. Figure 7.2 shows, for example, a 3x3 stiffness matrix. Elements A[1][1] = 10 pa.m = 10 N/m; A[2][3] = 7 $pa.m^2$ = 7 N, and A[3][3] = 13 $pa.m^2 * m$ = 13 N.m. This strategy for storing units not only requires much less memory than complete element-by-element storage of units, but it reflects the reality that most engineering matrices are in fact, convenient representations of equations of motion and equlibrium. The units of individual terms in these equations must be consistent.

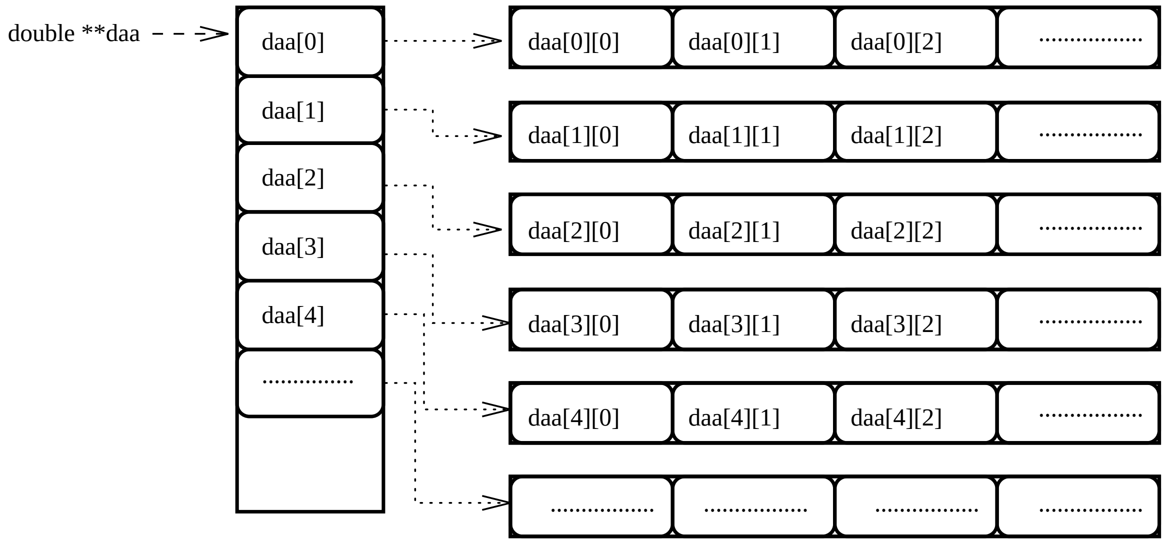


Figure 7.3: Matrix body for Indirect Storage Pattern

Matrices generated through the command language and file input are stored with an `INDIRECT` storage pattern. Details of the latter storage pattern are shown in Figure 7.3.

### 7.3.1 Skyline Matrix Storage

The method of skyline storage is suitable for large symmetric matrices, which are sparsely populated by non-zero elements along (or close to) the matrix diagonal. ALADDIN uses skyline storage for matrices generated during finite element analysis. To illustrate the general idea of skyline storage, consider symmetric matrix A given by

$$A = \begin{bmatrix} 10 & 4 & 0 & 0 & 3 & 0 \\ 4 & 45 & 0 & 1 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 6 & 3 \\ 3 & 0 & 0 & 6 & 4 & 34 \\ 0 & 0 & 0 & 3 & 34 & 20 \end{bmatrix}$$

We first note that indirect storage of A would require a $(6 \times 1)$ array of pointers, plus six one-dimensional arrays, each containing 6 elements of data type double. On an engineering

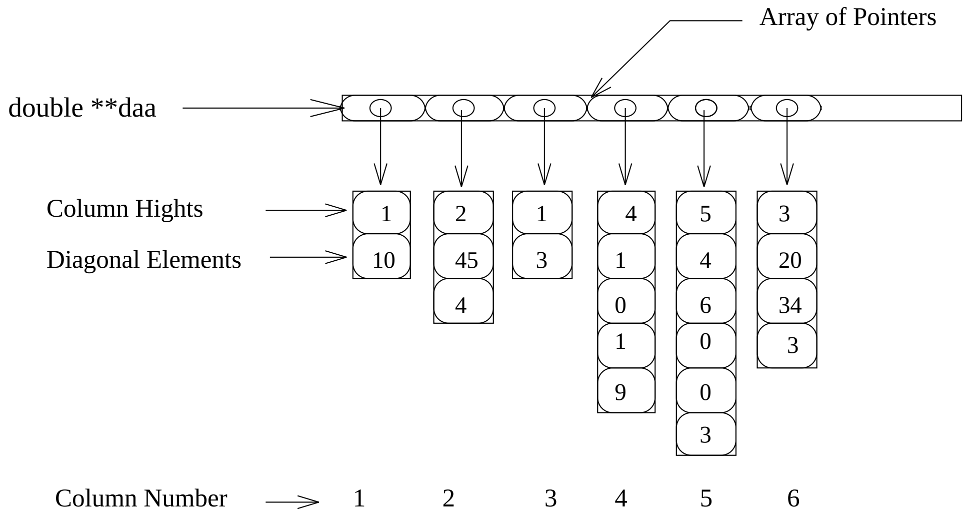


Figure 7.4: Method of Skyline Storage for Symmetric Matrix

workstation where pointers are 4 bytes, 312 bytes are needed for the matrix body. The method of skyline storage reduces memory requirements by taking advantage of matrix symmetries, and storing in each column, only those matrix elements from the matrix diagonal to the uppermost non-zero element. The "distance" between the diagonal and the last non-zero element is called the "column height." Figure 7.4 shows the layout of memory for skyline storage of matrix A. In each column, memory is allocated for the memory for the size of "column height" plus one, with the height stored in the first element of the corresponding array. The second component of the column array stores the matrix diagonal, and the last element, the upper-most non-zero element in the array. The column height is needed for computation of matrix operations – for example, matrix element (`iRowNo`, `iColumnNo`) is accessed and printed with the formula

```
ik = MIN( iRowNo, iColumnNo );
im = MAX( iRowNo, iColumnNo );

if((im-ik+1) <= spA->uMatrix.daa[im-1][0])
    printf(" %12.5e ", spA->uMatrix.daa[ im-1 ][ im-ik+1 ]);
else
    printf(" %12.5e ", 0.0);
```

Here `MIN()` and `MAX()` are macros that return the minimum and maximum of two input arguments, respectively.

**Functions in Matrix Library :** Table 7.2 contains a summary of matrix functions provided by ALADDIN. During matrix operations, consistency of units is checked, as is compatibility of matrix dimensions. Readers should notice the subtle difference between function pairs MatrixAdd() and MatrixAddReplace(), MatrixSub() and MatrixSubReplace(), and MatrixMult() and MatrixMultReplace(). In the functions MatrixInverseIteration() and MatrixHouseHolder(), matrices A and B must be symmetric.

| Matrix Function and Purpose | |
|---|---|
| Function | Purpose of Function |
| MatrixAlloc() | Allocate memory for matrix |
| MatrixFree() | Free memory for matrix |
| MatrixPrint() | Print content of matrix |
| MatrixCopy() | Make a copy of matrix |
| MatrixTranspose() | Compute transpose of matrix |
| MatrixAdd() | Compute sum of two matrices $[C] = [A] + [B]$ |
| MatrixSub() | Compute difference of two matrices $[C] = [A] - [B]$ |
| MatrixMult() | Compute product of matrices |
| MatrixNegate() | Negate matrix [B] = -[A] |
| MatrixAddReplace() | Replacement sum of two matrices $[A] = [A] + [B]$ |
| MatrixSubReplace() | Replacement difference of two matrices $[A] = [A] - [B]$ |
| MatrixMultReplace() | Replacement product of two matrices $[A] = [A].[B]$ |
| MatrixNegateReplace() | Negate matrix replacement [A] = -[A] |
| MatrixSubstitute() | Substitute small matrix into larger matrix |
| MatrixExtract() | Extract small matrix from larger matrix |
| MatrixSolve() | Solve linear equations $[A]\{x\} = \{b\}$ |
| MatrixInverse() | Compute matrix inverse $[A]^{-1}$ |
| MatrixInverseIteration() | Use inverse iteration to solve symmetric eigenvalue problem $[A]\{x\} = \lambda[B]\{x\}$ for lowest eigenvalue and eigenvector |
| MatrixHouseHolder() | Use Householder transformation and QL Algorithm to solve $[A]\{x\} = \lambda\{x\}$ |

Table 7.2: **Selected Matrix Operations**

### 7.3.2 Units Buffers for Matrix Multiplication

The handling of units in multiplication of two dimensional matrices needs special attention. Let A be a $(p \times q)$ matrix with row units buffer [ $a_1, a_2, \cdots, a_p$ ] and column units buffer[ $b_1, b_2, \cdots, b_r$]. And let B be a $(q \times r)$ matrix with row units buffer [ $c_1, c_2, \cdots, c_q$ ] and a column units buffer $[d_1, d_2, \cdots, d_q]$. The units for elements $(A)_{ik}$ and $(B)_{kj}$ are $a_i * b_k$ and $c_k * d_j$, respectively. Moreover, let C be the product of A and B. From basic linear algebra we know that $(C)_{ij} = A_{ik} * B_{kj}$, with summation implied on indice `k`. The units accompanying $(C)_{ij}$ are $a_i b_k * c_k d_j$ for k = 1, 2, ..., q. Due to the consistency condition, all of the terms in $\sum_{k=1}^{q} A_{ik} * B_{kj}$ must have same units. Put another way, the exponents of units must satisfy the product constraint $a_i b_1 c_1 d_j = a_i b_2 c_2 d_j = \cdots = a_i b_q c_q d_j$. The units for $C_{ij}$ are $a_i b_1 c_1 d_j$.

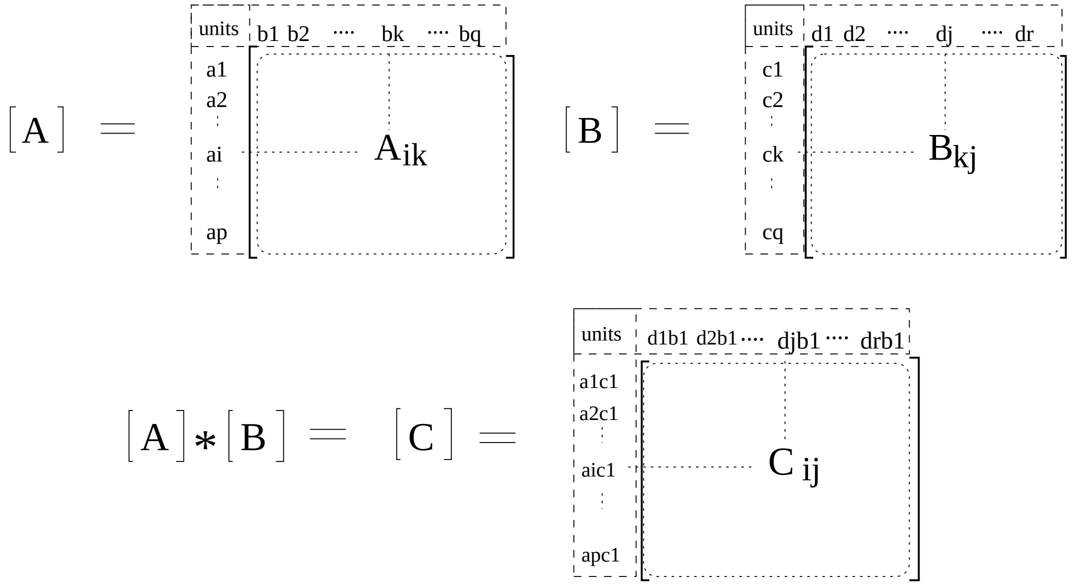


Figure 7.5: Units Buffer Multiplication of Two Matrices

The units buffers for matrix C are written as a row units buffer [ $a_1c_1, a_2c_1, \cdots, a_pc_1$], and a column buffer is [ $d_1b_1, d_2b_1, \cdots, d_rb_1$]. This arrangement of units exponents is graphically displayed in Figure 7.5. It is important to notice that although the units for C matrix are unique, the solution for the units buffers is not unique.

### 7.3.3 Units Buffers for Inverse Matrix

The units buffers for an inverse matrix may be obtained as the special case of above results. Again let A be the square matrix of interest, with number of rows and columns equal (i.e. p = q). Let matrix B be the inverse matrix of matrix A. From the previous section we know that the units for element $C_{ij}$ are $a_i b_k c_k d_j$, and because B is the inverse of matrix A, the diagonal elements of the matrix C must be dimensionless (i.e.

$a_i b_k c_k d_i = 1$ for $k = 1, 2 \cdots p$ and $i = 1, 2 \cdots p$). However, because contents of the units buffers need not be unique, and

$$c_k d_i = \frac{1}{a_i b_k}$$

we can arbitrarily select $c_k$ and $d_j$ with the following set as the units buffers:

$$c_k = \frac{1}{b_k}$$

for $k = 1, 2, \cdots p$, and $d_i = 1/a_i$ for $i = 1, 2, \cdots p$. The reader should notice that in this arrangement of units exponents, the row units buffer of inverse matrix of A is the inverse of A's column units buffer, and the column buffer of inverse matrix of A is the inverse if A's row units buffer.

Let [M] be a $(n \times n)$ square matrix, $[M]^{-1}$ it's inverse. In expanded matrix form (i.e. including units buffers), [M] and $[M]^{-1}$ can be written:

$$[M] = \begin{bmatrix} units & b_1 & b_2 & \cdots & b_n \\ a_1 & M_{11} & M_{12} & \cdots & M_{1n} \\ a_2 & M_{21} & M_{22} & \cdots & M_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_n & M_{31} & M_{32} & \cdots & M_{nn} \end{bmatrix}$$

$$[M]^{-1} = \begin{bmatrix} units & {a_1}^{-1} & {a_2}^{-1} & \cdots & {a_n}^{-1} \\ {b_1}^{-1} & M^*_{11} & M^*_{12} & \cdots & M^*_{1n} \\ {b_2}^{-1} & M^*_{21} & M^*_{22} & \cdots & M^*_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ {b_n}^{-1} & M^*_{31} & M^*_{32} & \cdots & M^*_{nn} \end{bmatrix}$$

The corresponding [IR] and [IL] are :

$$[IR] = [M] \times [M]^{-1} = \begin{bmatrix} units & b_1.{a_1}^{-1} & b_1.{a_2}^{-1} & \cdots & b_1.{a_n}^{-1} \\ a_1.{b_1}^{-1} & 1 & 0 & \cdots & 0 \\ a_2.{b_1}^{-1} & 0 & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_n.{b_1}^{-1} & 0 & 0 & \cdots & 1 \end{bmatrix}$$

$$= \begin{bmatrix} units & {a_1}^{-1} & {a_2}^{-1} & \cdots & {a_n}^{-1} \\ a_1 & 1 & 0 & \cdots & 0 \\ a_2 & 0 & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_n & 0 & 0 & \cdots & 1 \end{bmatrix}$$

$$[IL] = [M]^{-1} \times [M] = \begin{bmatrix} units & a_1 & a_2 & \cdots & a_n \\ {a_1}^{-1} & 1 & 0 & \cdots & 0 \\ {a_2}^{-1} & 0 & 0 & \cdots & 0 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ {a_n}^{-1} & 0 & 0 & \cdots & 1 \end{bmatrix}$$

We observe that the units of [IL] and [IR] depend only on the row units buffer of the matrix [M].

**Example from Structural Engineering :** In undergraduate structural analysis classes we learn that an external force vector {f} can be written as the product of a stiffness matrix `K` and displacement vector `d`.

$$[K]\{d\} = \{f\}$$

Suppose that we are working in a three dimensional coordinate space (x,y,z). Without a loss of generality in discussion, we can limit the total number of degrees of freedom to three; two translations $u$ and $v$ in directions x and y, respectively, and a rotational degree of freedom $\theta$ about the z axis, The translational forces and moment in vector {f} are $f_1, f_2$, and $M$. In expanded form the abovementioned equation is:

$$\begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix} \times \begin{Bmatrix} u \\ v \\ \theta \end{Bmatrix} = \begin{Bmatrix} f_1 \\ f_2 \\ M \end{Bmatrix}$$

Because $u$ and $v$ are translational displacements, they will have length units, "m" or "in." External forces $f_1$ and $f_2$ will have units newton "N," or pound force "lbf". For the underlying equations of equilibrium to be consistent, units for $k_{11}, k_{12}, k_{21}$ and $k_{22}$ must be "N/m" or "lbf/in." And because $\theta$ has non-dimensional radian units, $k_{13}$ and $k_{23}$ have units "N" or "lbf." The third equation represents rotational equilibrium, with its right-hand side having units of moment (i.e "N.m" or "lbf.in"). The units for $k_{31}$ and $k_{32}$ are "N" or "lbf," and units for $k_{33}$, "N.m" or "lbf.in." For SI units we write:

$$[K] = \begin{bmatrix} units & N/m & N/m & N \\ & k_{11} & k_{12} & k_{13} \\ & k_{21} & k_{22} & k_{23} \\ m & k_{31} & k_{32} & k_{33} \end{bmatrix}$$

and for US units:

$$[K] = \begin{bmatrix} units & lbf/in & lbf/in & lbf \\ & k_{11} & k_{12} & k_{13} \\ & k_{21} & k_{22} & k_{23} \\ in & k_{31} & k_{32} & k_{33} \end{bmatrix}.$$

The equations of equilibrium and units buffers for the corresponding compliance matrix are:

$$\begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} \times \begin{Bmatrix} f_1 \\ f_2 \\ M \end{Bmatrix} = \begin{Bmatrix} u \\ v \\ \theta \end{Bmatrix}$$

For SI units,

$$[C] = \left[ \begin{array}{rccr} units & & & 1/m \\ m/N & c_{11} & c_{12} & c_{13} \\ m/N & c_{21} & c_{22} & c_{23} \\ 1/N & c_{31} & c_{32} & c_{33} \end{array} \right]$$

A similar matrix C can be written for US units. We observe that the units buffers for the compliance matrix, which is the inverse of the stiffness matrix, follows the general rule stated at the conclusion of the previous section.

Now let's look at the numerical details and units of the matrix product $[K] \times [C]$ – for notational convenience, we will call the identity product [IR]. The elements first and second rows of $[IR]$ have units of [ 1, 1, 1/m], and the third row of $[IR]$, units [m,m,1]. In expanded matrix form, we have:

$$[IR] = [K] \times [C] = \left[ \begin{array}{rccr} units & & & 1/m \\ & 1 & 0 & 0 \\ & 0 & 1 & 0 \\ m & 0 & 0 & 1 \end{array} \right]$$

Similarly we have

$$[IL] = [C] \times [K] = \left[ \begin{array}{rccc} units & & & m \\ & 1 & 0 & 0 \\ & 0 & 1 & 0 \\ 1/m & 0 & 0 & 1 \end{array} \right]$$

Three points are noted about this result. Matrices [IL] and [IR] are not equal as most engineers might expect – instead, [IL] is the transpose of [IR]. Second, [IL] and [IR] are not symmetric with respect to the units or units buffers. Finally, these "identity" matrices are not dimensionless. Even though the values of off-diagonal elements are all zero, some of them have non-dimensional units.

Some readers may feel uncomfortable knowing that so-called "indentity" matrices are not dimensionless, and may wonder how could that be. Actually, when you look carefully into the equations, the results are not surprising. For example, let's take the first column of compliance matrix [C] and label it $\{C_1\} = [c_{11}, c_{21}, c_{31}]^T$. When [K] $\{C_1\}$

is compared to [K] $\{d\}$, one quickly realizes that $[c_{11}, c_{21}, c_{31}]$ takes the place of [u, v, $\theta$] (here $c_{11}, c_{21}$ are the displacement per unit force, and $c_{31}$ is the rotation per unit moment). Given that the units for {f} are $[force, force, moment]^T$, then units for the first column of matrix [IR] will be [force/force, force/force, moment/force] (or [1, 1, m]). Similar results may be derived for the remaining columns of C, and in fact, may be generalized.

# Chapter 8

# Architecture and Design of ALADDIN

## 8.1 Introduction

A key design objective for ALADDIN is the development of a program structure that is very modular. We have captured this principle by designing the program architecture, and supporting software modules, around the compiler construction tool called YACC (an acronym for Yet Another Compiler Compiler) [16]. YACC takes the language description and automatically generates C code for a parser that will match streams of input against the rules of the language. A key advantage in using YACC is that it provided us with the freedom to experiment with the language design/semantics, and to add and delete new features. Experience in other domains (i.e. computer science) indicates such strategies result in modular software that is extensible, and easier to maintain.

## 8.2 Program Modules and Key Data Structures

Figure 8.1 shows the interfaces and relationships among the seven modules in ALADDIN. They are (a) Main Module, (b) Pre-Processor Module, (c) Central Control Module, (d) Finite Element Base Module, (e) Element Library Module, (f) Matrix Module, and (g) an Engineering Units Module. A detailed summary is as follows:

**Main Module :** The main module acts as the entry point for program execution. It loads information specificed in ALADDIN's header files into the symbol table, directs the source of expected input (i.e. keyboard or files), and details of command options, decides whether or not to check the input syntax errors with/without execution of commands. The main module calls the pre-processor module, and executes the program.

**Preprocessor Module :** ALADDIN's preprocessor module parses input streams from either the keyboard of file, and prepares an array of machine instructions that will be executed by ALADDIN's stack machine. The stack machine is part of the program control

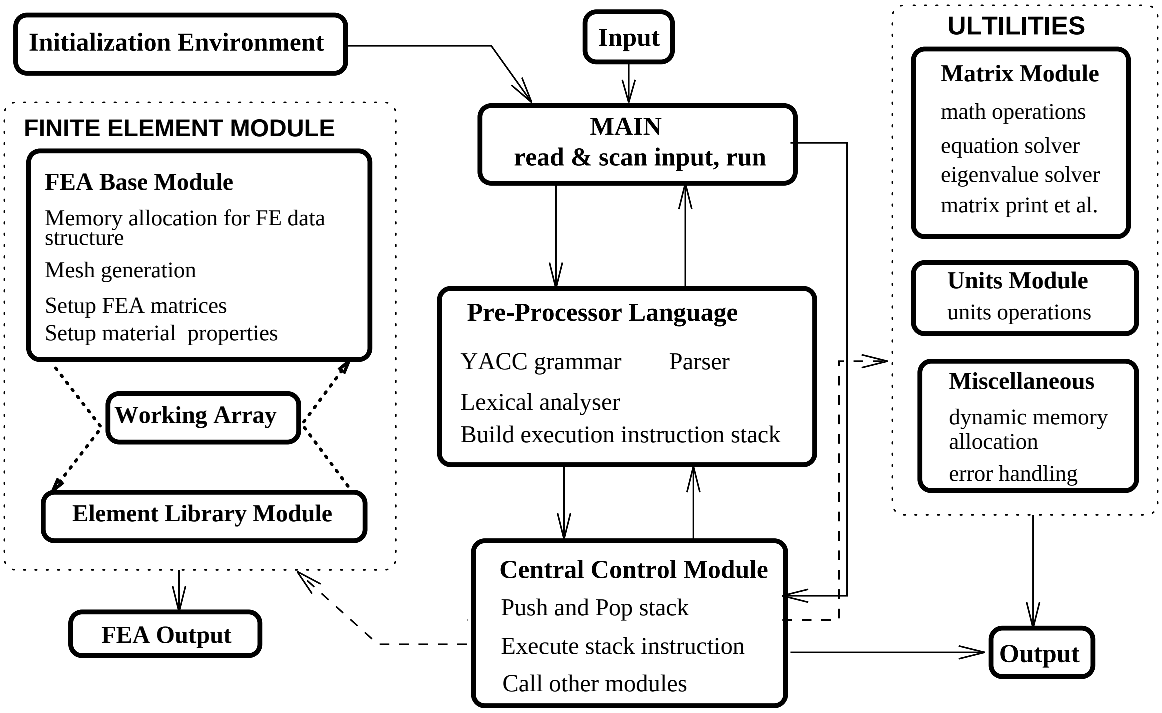


Figure 8.1: Architecture of Program

module.

**Program Control Module :** The central control module is composed of a stack machine, and is the heart of ALADDIN. The design and implementation details for the stack machine will be described in the next section of this report.

**Finite Element Base Module :** The finite element base module contains ALADDIN's framework for finite element analysis. This module has code to generate nodes and elements in finite element meshes, a problem descriptor for boundary and initial conditions, material and sections properties, and facilities to setup external loads. It contains functions to allocate and compute finite element mass, stiffness, and external load matrices.

The heart of the finite element base structure is the data structure `frame`. It has been designed to read, store and update information in finite element analysis. "frame" comprises of a set of control parameters and pointers to node, element, material and load, which include the basic input formation.

```
typedef struct frame {
char                *name; /* Title */
        int              no_nodes; /* no of nodes                              */
        int              no_rigid; /* no of rigid body elements                */
        int           no_elements; /* no of flexible elements                  */
        int       no_element_attr; /* no of element attributes                 */
        int         no_node_loads; /* no of nodal loads                        */
        int      no_element_loads; /* no of element loads                      */
        int              no_dimen; /* no of dimension                          */
```

```
        int               no_dof; /* no of dof per node--in global           */
        int     no_nodes_per_elmt; /* max no of nodes per element             */
        NODE               *node; /* array of nodes                          */
        RIGID             *rigid; /* rigid body information                  */
        ELEMENT         *element; /* array of elements                       */
        ELEMENT_ATTR       *eattr; /* array of element attributes             */
        NODE_LOADS      *nforces; /* array of nodal forces                   */
        ELEMENT_LOADS   *eforces; /* array of element forces                 */
        LOAD_TYPE     *load_type; /* test for dynamic/static problems           */
        LOAD_HISTORY  *load_hist; /* load_history structure for dynamic analysis */
        LOAD_STEP     *load_step; /* load_step structure for nonlinear analysis  */
        TIME               *time; /* time data structure for dynamic analysis    */
        MATRIX       *eigenvalue; /* Eigenvalue vector */
} EFRAME;
```

The control parameters define the problem size and the extend of allocation. These are global parameters and keeps constant throughout the analysis. The node data structure, NODE, contains the information about the nodes coordinates, boundary constraint information of that node, the initial displacement and velocity of the nodes. And the element data structure, ELEMENT, contains the information of its connection to the nodes in the element, the relation between the global and local degree of freedoms, material and geometry of the element et al. They are given as:

```
typedef struct node {
        QUANTITY      *coord;     /* Coordinates of node            */
        int           *bound_id;  /* Boundary Constraint ID        */
        int            rb_num;    /* component of rigid body number */
        QUANTITY      *disp;
        QUANTITY      *velocity;
        MATRIX        *TrT;
} NODE;

typedef struct element_state {
        int             state; /* state = 1    elastic-plastic deformation */
                               /* state = 0    elastic deformation         */
} ELEMENT_STATE;

typedef struct element_response {
     MATRIX              *Forces;
     MATRIX              *stress;
     MATRIX               *displ;
     MATRIX            *velocity;
     MATRIX           *strain_pl;
     double      *effect_pl_strain;
     QUANTITY          min_moment;
     QUANTITY      max_moment, Mzc;
     QUANTITY  min_shear, max_shear;
} RESPONSE;

typedef struct element {
        char     *elmt_attr_name;  /* Elmt_Attr name                        */
        int        *node_connect;  /* List of Nodal Connectivities          */
```

```
        int          elmt_attr_no;  /* Elmt Attribute No (stored in frame) */
        int              *d_array;  /* Destination array storage           */
        RESPONSE              *rp;  /* Response Pointer                    */
        ELEMENT_STATE        *esp;  /* Deformation State of the Element    */
        MATER_LOAD_CURVE *LC_ptr;   /* properties for describe yield       */
                                    /* surface and stress-strain curve     */

} ELEMENT;
```

The element attribute information is included in the ELEMENT_ATTR data structure.

```
typedef struct elmt_attr {
        char                  *name;
        char             *elmt_type;
        char              *material;
        char               *section;
        int        *map_ldof_to_gdof;
        QUANTITY      *work_material;   /* Working Array for Material Properties */
        QUANTITY       *work_section;   /* Working Array for Section  Properties */
} ELEMENT_ATTR;
```

There are two working arrays in the element attribute data structure. One for the material properties and the other for the section property. For given element no, the material and section properties can be found directly from these working array instead of looking into the hashing table.

**Finite Element Library Module :** The finite element library module is a collection of finite felement functions. Like FEAP, ALADDIN allows engineers to add new finite elements to the program without having to interact other parts of the program. At the time of writing (December 1994), the library contains two- and three- dimensional beam/column elements, a plane stress/strain element, and two four node shell elements. Each element has code to compute stiffness and mass matrices, plus for a given displacement vector, a vector of internal forces acting on the nodal degrees of freedom.

**Matrix Module :** The matrix module contains functions to allocate and deallocate memory for matrices, to compute basic operations of matrices, to compute solutions to families of linear equations, and to solve the symmetric eigenvalue problem.

**Units Module :** The units module provides the units operations of engineering quantities and matrices for both US and SI systems of units. Operations for units conversion are provided, as are facilities to turn units operations on/off. The following data structure stores details of the units name, conversion factor, length, mass, and time exponents:

```
   static struct {
         char         *name;     /* units name   */
         double  conversion;     /* scale factor */
         int   length_expnt;
         int     mass_expnt;
         int     time_expnt;
         int     temp_expnt;
```

```
        int     units_type;
   } eng_units[] = {
             "micron",   1E-6,  1,  0,  0, 0, SI,    /* Units of Length */
                 "mm", 0.0010,  1,  0,  0, 0, SI,
                 "cm", 0.0100,  1,  0,  0, 0, SI,
                 "dm", 0.1000,  1,  0,  0, 0, SI,
                  "m",      1,  1,  0,  0, 0, SI,
                 "km",   1000,  1,  0,  0, 0, SI,
                  "g", 0.0010,  0,  1,  0, 0, SI,    /* Units of Mass */
                 "kg",      1,  0,  1,  0, 0, SI,
                 "Mg",   1E+6,  0,  1,  0, 0, SI,
                "sec",      1,  0,  0,  1, 0, SI_US, /* Units of Time */
                "min",     60,  0,  0,  1, 0, SI_US,
                 "hr",   3600,  0,  0,  1, 0, SI_US,
              "deg_C",      1,  0,  0,  0, 1, SI,    /* Temperature */
                  "N",      1,  1,  1, -2, 0, SI,    /* Units of Force */
                 "kN",   1000,  1,  1, -2, 0, SI,
                "kgf",  9.807,  1,  1, -2, 0, SI,
                 "Pa",      1, -1,  1, -2, 0, SI,    /* Units of Pressure */
                "kPa",   1000, -1,  1, -2, 0, SI,
                "MPa",   1E+6, -1,  1, -2, 0, SI,
                "GPa",   1E+9, -1,  1, -2, 0, SI,
                "Jou",      1,  2,  1, -2, 0, SI,    /* Energy Units */
                 "kJ",   1E+3,  2,  1, -2, 0, SI,
               "Watt",      1,  2,  1, -3, 0, SI,    /* Power Units */
                 "kW",   1000,  2,  1, -3, 0, SI,
    }
```

In the SI units system, for example, we employ the character "N" to represent one Newton force, and the character "m" to represent one meter length. Both units have a scale_factor equal to one – it follows that the reference unit for moment would be "N*m", with a scale_factor also equal to 1.0.

**Initialization Module :** ALADDIN employs a linked list symbol table data structure with operations for loading, storing, and retrieving information during a matrix/finite element analysis. Items in the symbol table are required to have unique names. The heart of the symbol table is defined by the data structure:

```
/* Data Structure for Node in Symbol Table */

typedef struct Symbtab_element {
   char *cpSymName;
   short      type;
   union {
      double                  value;   /* Store a number             */
      char                     *str;   /* String                     */
      int                (*defn)();   /* Function, Procedure        */
      void            (*voidptr)();   /* Built-in Math/FE Functions  */
      double        (*doubleptr)();   /* Built-in Math/FE Functions  */
      MATRIX       *(*matrixptr)();   /* Built-in Matrix Functions   */
      QUANTITY    *(*quantityptr)();   /* Built-in Quantity Functions */
      ARRAY         *(*elmt_ptr)();   /* Elmt Library ptr to func    */
```

```
      QUANTITY                    *q;   /* Engineering Quantity        */
      MATRIX                      *m;   /* Matrix                      */
      DIMENSIONS              *dimen;   /* Basic Dimension             */
      ELEMENT_ATTR              *eap;   /* Element Attribute ptr       */
      SECTION_ATTR              *sap;   /* Section Attribute ptr       */
      MATERIAL_ATTR             *map;   /* Materials Attribute ptr     */
   } u;
   struct  Symbtab_element   *next;
 } SYMBOL;
```

Here, MATRIX, QUANTITY and DIMENSIONS are the data structures for matrix, quantity and units, defined in Chapter 2. The names ARRAY, ELEMENT_ATTR, SECTION_ATTR and MATERIAL_ATTR are pointers to the element, section and material attributes in finite element analysis.

The initialization environment loads keywords, constants, finite element analysis information (e.g. material and section properties), and built-in functions needed for the command language into ALADDIN's symbol table. We use the returned data type, and details of the argument list (i.e. number and types of data), to organize built-in functions into six groups. They are:

[1] Functions requiring one matrix argument, and returning a quantity. Two examples are L2Norm() and QuanCast().

[2] Functions requiring no argument, and return a matrix. Two examples from the finite element section are Stiff() and Mass().

[3] Functions requiring one matrix argument, and returning a matrix. Two examples are Copy() and Trans().

[4] Functions requiring two matrix arguments, and returning a matrix. One example is Solve(,).

[5] Functions that accept a variable number of matrix arguments, and return a matrix.

[6] Special functions for the generation of finite element meshes.

The following script of code shows how functions with one matrix argument are defined in a data structure called builtin1_matrix[], and subsequently installed in ALADDIN's symbol table by the function Init_Problem().

```
 /* [a] : Data structure for built-in functions */

    static struct {
           char           *name;
           MATRIX    *(*func)();
    } builtin1_matrix[] = {
          "Copy",  MatrixCopy,
         "Trans",  MatrixTranspose,
```

```
   };

 /* [b] : Init_Problem() : Load builtin functions into Symbol Table */

   Init_Problem()
   {
   int          i;
   SYMBOL_PTR spA;

     /* [a] : Load builtin functions into symbol table */

        for (i = 0; i < (sizeof(builtin1_matrix)/sizeof(builtin1_matrix[0])); i++)  {
             spA = build_table(builtin1_matrix[i].name, BLTIN1_MATRIX, 0.0);
             spA->u.matrixptr = builtin1_matrix[i].func;
        }
   }
```

Here we use the C function `build_table()` to install the function names into the symbol table. Pointers to the base address of the function are assigned with the statement `spA->u.matrixptr = builtin1_matrix[i].func`.

**Material and Section Property Data Structures :** Material and section properties are stored in the material and section attribute data structures: MATERIAL_ATTR, SECTION_ATTR. They are given as:

```
 typedef struct materials {
        char                  *name;    /* Material name                   */
        QUANTITY                  E;    /* Young's modulus                 */
        QUANTITY                  G;    /* Shear modulus                   */
        QUANTITY                 fy;    /* Yield stress                    */
        QUANTITY                 fu;    /* Ultimate stress                 */
        QUANTITY                 ET;    /* Tangent Young's Modulus         */
        double                   nu;    /* Poission's ratio                */
        QUANTITY            density;
        QUANTITY     *alpha_thermal;    /* thermal expansion coefficient   */
        MATER_LOAD_CURVE    *LC_ptr;    /* parameters for describing yield  */
                                        /* surface and stress strain curve */
      } MATERIAL_ATTR;

typedef struct section_attr {
        char *section_name;
        int   section_type;
        QUANTITY   Ixx, Iyy, Izz;
        QUANTITY   Ixz, Ixy, Iyz;
        QUANTITY   weight;                  /* Section weight */
        QUANTITY   bf;                      /* Width of flange        */
        QUANTITY   tf;                      /* thickness of flange    */
        QUANTITY   depth;                   /* Section depth          */
        QUANTITY   area;
        QUANTITY   plate_thickness;
        QUANTITY   tor_const;               /* Torsional Constant J   */
        QUANTITY   x_coord, y_coord, z_coord /* centroid coord        */
```

```
    QUANTITY   rT;                          /* Section Radius of gyration*/
    QUANTITY   width;                       /* Section width          */
    QUANTITY   tw;                          /* Thickness of web       */
  } SECTION_ATTR;
```

In many of above data structures, MATER_LOAD_CURVE data structure is used to describe the stress-strain curve of the material for each element.

```
typedef struct mater_load_curve {
   char                 *name;   /* load curve type                      */
   double                  *R;   /* radius of yield surface              */
   double        **back_stress;  /* back stress vector                   */
   double                  *H;   /* tangent of stress-plastic strain curve*/
   double               alpha;   /* parameter for Ramberg-Osgood relation */
   double                   n;   /* strain hardening exponent            */
   double                beta;   /* parameter for strain hardening       */
                                 /* beta = 0 kinematic hardening         */
                                 /* beta = 1 isotropic hardening         */
} MATER_LOAD_CURVE;
```

Many material properties, such as `R`, the radius of yield surface in $\pi$-plane and `H`, the tangent of stress-plastic strain curve, are point dependent. Therefore, arrays are used to identify the corresponding properties for every integration points. The character string `name` identifies the type of the stress-strain curve. Two stress strain curves are currently available. They are Ramberg-Osgood stress strain curve, and the Bi-Linear stress strain curve.

**Working Array :** The `P_ARRAY` data structure transfers information between finite element base module and element library module. This information is used to compute element level mass and stiffness matrices, and internal force vectors, for both linar and nonlinear problems.

## 8.3 Design and Implementation of Stack Machine

ALADDIN employs a finite-state stack machine model, which follows in the spirit of work presented by Kernighan and Pike [18]. Stack machines are suitable for modeling systems that have a finite number of internal configurations or states, and whose behavior evolves as a linear sequence of discrete points in time. ALADDIN's stack machine reads blocks of command statements from either a problem description file, or the keyboard, converts them into stack machine instructions, and finally, executes the statements.

ALADDIN employs three connected data structures – an array of machine instructions, a program stack, and a symbol table – plus carefully designed algorithms and software to handle the processing of input statements, and blocks of input statements. Together these data structures and algorithms form the heart of ALADDIN's stack machine. The stack machine processes blocks of input statements in two steps:

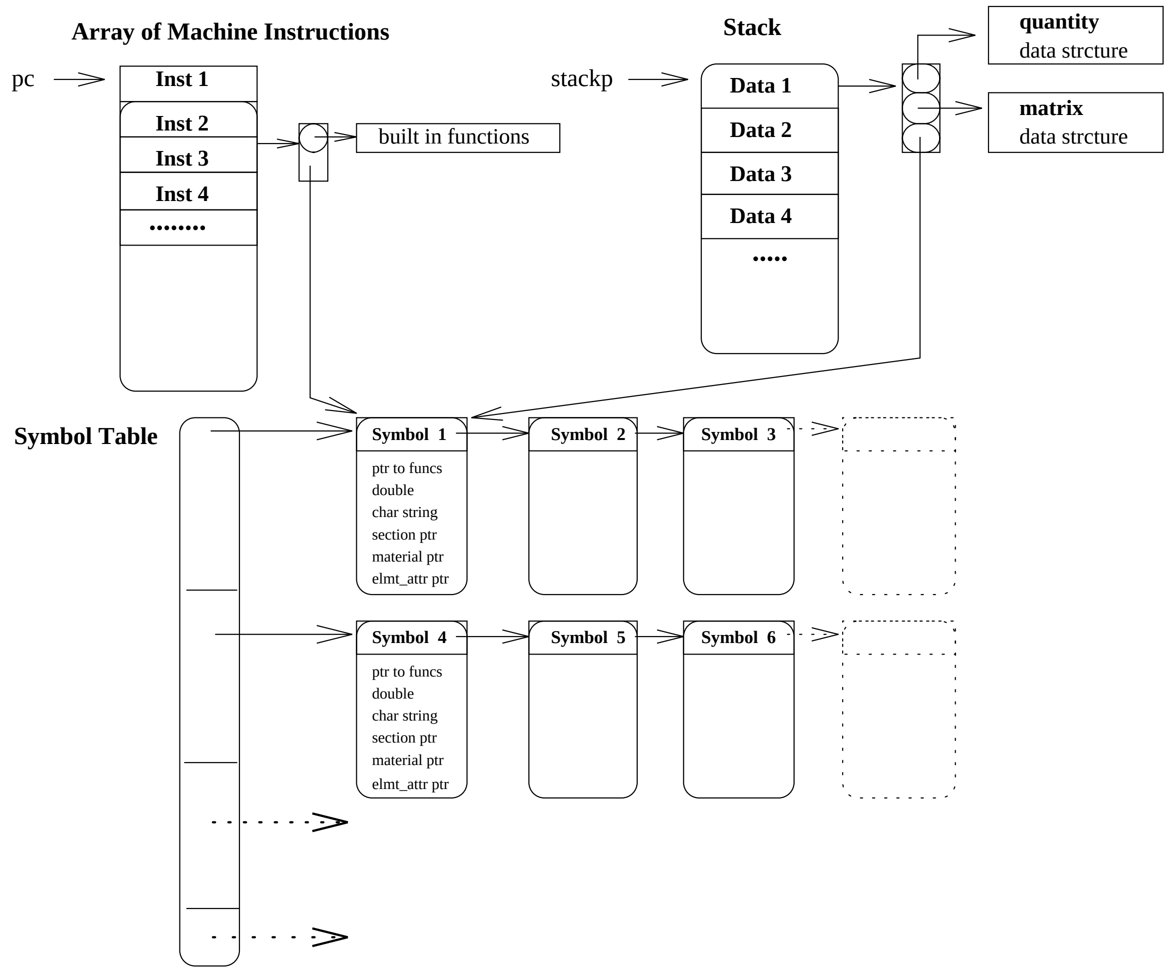


Figure 8.2: Data Structures in ALADDIN's Stack Machine

[1] Parse the input statement (or block of statements), and construct an array of machine instructions. Groups of machine instructions are composed of calls to functions, plus instructions to push and pop data and operands to/from the program stack.

[2] Step through the array of machine instructions, and execute the functions pointed to by the machine instructions.

Figure 8.2 is a simplified schematic of the stack machines participating data structures, and their connectivity.

**Array of Machine Instructions :** The array of machine instructions and program stack are declared with the following C statements:

```
/* Define array of machine instructions */

typedef int (*Inst)();
#define STOP (Inst) 0;
```

```
#define NPROG 5000;

Inst prog [ NPROG ];
Inst *progp;
Inst *pc;

/* Define data structure for program stack */

typedef union Datum  {
        QUANTITY    *q;
        MATRIX      *m;
        SYMBOL    *sym;
} DATUM;

#define NSTACK 500;
static DATUM stack [ NSTACK ];
```

The C statement `Inst prog [ 5000 ]` allocates memory for the array of machine instructions. Each element of `prog` is a pointer of type int. Variables `*progp` and `*pc` are pointers of type `Inst`, which will be used to track elements in the array of instructions.

The data structure DATUM is a union of pointers to quantity, matrix and symbol table items. Each element of the program stack is either a pointer to a quantity, a pointer to a matrix, or a pointer to a symbol table entry. The C statement `static DATUM [ 500 ]` allocates a block of memory for the program stack containing 500 elements of structure `DATUM`.

The C function `Code()` plays a central role in the assembly of machine instructions. For example, the function call `Code(Push);` will add to the machine array, an element containing a pointer to function `Push()`, and then increments the program pointer `progp`.

```
/*
 *  ====================================================================
 *  The function code() builds up the machine stack prog, adding data
 *  types Inst to it. The arguments to code() are pointers
 *  to the functions defined later on in code.c, and pointers to
 *  data types SYMBOL, which have been coerced by the use of the
 *  cast Inst.
 *  ====================================================================
 */

Inst *Code( Inst f )
{
Inst *oprogp = progp;

   if(progp >= &prog[NPROG])
      ExecutionError("ERROR >> program too big", (char *) 0);

    *progp++ = f;

    return oprogp;
}
```

```
/*
 *  ======================================================================
 *  Push DATUM d onto the stack, and increment stackp to next element.
 *  ======================================================================
 */

Push( DATUM d )
{
    if(stackp >= &stack[NSTACK])
       ExecutionError("ERROR >> stack overflow", (char *) 0);

    *stackp++ = d;  /* *stackp=d; stackp++; */
}
```

In functions `Code()` and `Push()`, `ExecutionError()` is a function that reports run-time execution errors in the interpretation of input commands (e.g overflows of the program stack). Function `Code()` has a single argument; the function call `Code (Push);` will, for example, add to the machine array, an instruction to call function `Push()`.

**Execution of Program Stack :** An execution pointer marches along the machine array and implements the instruction set assembled during step [1] of the input process. The machine instructions will push and pop items to/from the program stack, and call external C functions as directed by the arguments to `Code(...)`.

```
Execute(p)
Inst *p;
{
     for (pc = p; *pc = STOP; ) {
          pc = pc + 1;
          if( Check_Break() != 0) break;
          (*(*(pc-1)))();
     }
}
```

The code generated during parsing has been carefully arranged so that a `STOP` command terminates each sequence of instructions that should be handled by a single call to `Execute()`.

### 8.3.1 Example of Machine Stack Execution

We now demonstrate use of the stack machine by working step-by-step through the details of processing the assignment `x = 2 in;`. We assume that Step [1] of the input process is complete, with the array of machine instructions taking the values shown on the left-hand side of Figure 8.3. During Step [1] of the input process, the YACC parser – details to be described in the next section – recognizes that `x` is a variable with data type VAR, and `2 in` as a quantity with number `2` and a units dimension `in`. The grammatical rule for assignment (i.e. `"="`) has right associativity. Hence, when the machine is executed,

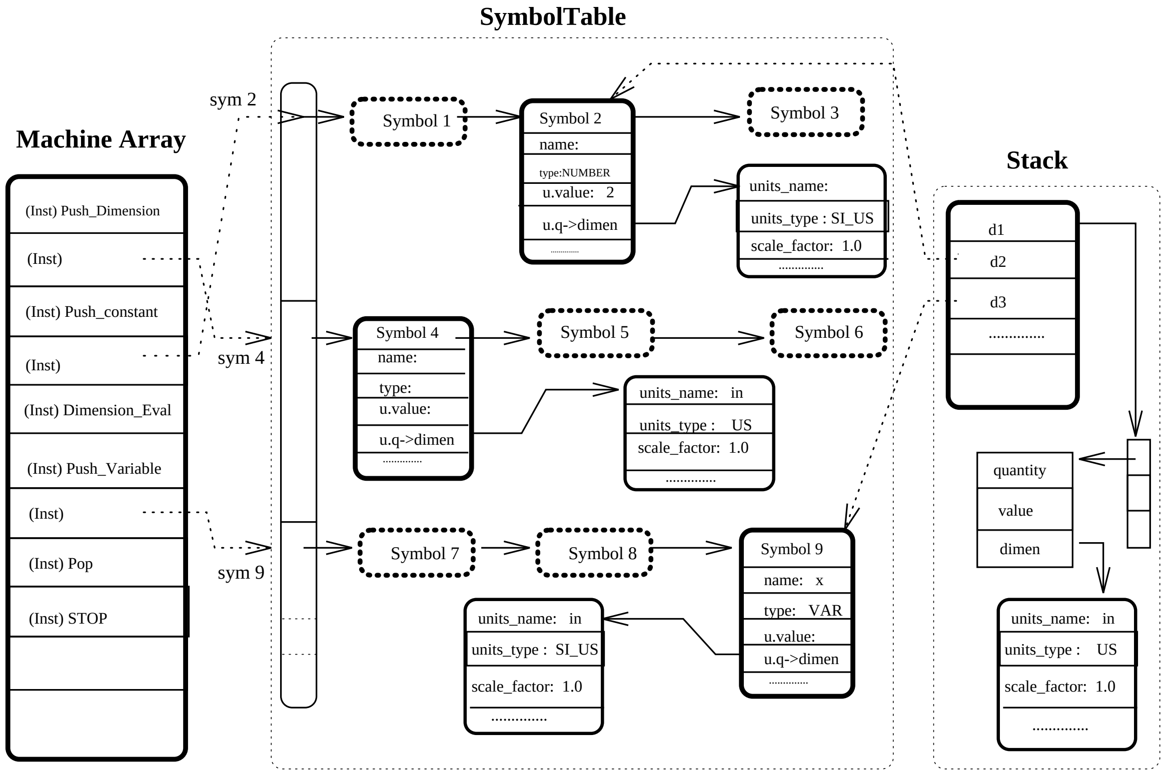


Figure 8.3: Schematic of Machine and Stack for x = 2 in

`2 in` will be pushed onto the stack first, and in two steps; first the unit `in` is pushed, and then the number `2`.

Detailed snapshots of the program operation for Step [2] of the processing are located at the end of this subsection. The step-by-step procedure for execution of the stack machine is:

[1] Push symbol table pointer onto stack for the variable `in`.

[2] Push a constant `2` onto the stack.

[3] Pop both `2` and `in` off the stack, and combine them into a single quantity `2 in`. The quantity is pushed back onto the stack.

[4] Push onto the stack, the symbol table pointer to variable `x`.

[5] Pop `x` and `2 in` from the stack. Assign `2 in` to `x` and push `x` back onto the stack.

[6] Data `x` is popped and cleared from the stack.

Figures 8.4 to 8.9 show the relevant details of the machine array, symbol table, and program stack for Steps [1] to [6], respectively. In each of these figures, `pc` is a program counter that points to elements in the array of machine instructions – the elements of the machine array are pointers to C functions. The dashed and solid arrows represent the two stages of the `pc` positions. As detailed in the C function `Execute()`, and shown in Figure 8.4, program counter `pc` initially points to `(Inst) Push_Dimension`. The position of `pc` is represented by the (dotted) left-to-right arrow in Figure 8.4. Notice, however, that `pc` is incremented to the (solid) arrow position (i.e `pc = pc+1`) before `Push_Dimension()` is called. Now (`pc-1`) points at `(Inst) Push_Dimension`. The details of `Push_Dimension()` are as follows:

```
/*
 *  ========================================================
 *  Push_Dimension() : push dimensioned quantity onto stack
 *  ========================================================
 */

void Push_Dimension()
{
DATUM    d;
int length;

    d.q = (QUANTITY *) MyCalloc(1,sizeof(QUANTITY));
    d.q->dimen = NULL;
    d.q->value = ((SYMBOL *) *pc)->u.q->value;

    pc = pc + 1;
    Push(d);
}
```

The machine instruction at the new value of `pc` points to a quantity stored in the symbol table. `Push_Dimension()` allocates memory of a new quantity (given by `d.q`), and copies to it, the contents of the symbol table entry. The result is pushed onto the program stack.

After unit `in` and number `2` are both pushed onto stack (i.e. `d1` and `d2` on the stack shown in Figure 8.3), function `Dimension_Eval()` pops `d1` and `d2` from the stack, assigns the units `in` to number `2`, and produces the quantity `2 in` back onto the stack.

Next, the symbol table pointer for variable `x` is pushed onto the stack by C function `Push_Variable()`. Details of this step are shown in Figure 8.7.

The C function `Assign_Quantity()` assigns the quantity `2 in` to variable `x`, and pushes the new `x` onto stack. Details of this step are shown in Figure 8.8. The second-to-last machine instruction pops the last item from the stack, as shown in Figure 8.9. Finally, `(Inst) STOP`, halts the looping mechanism in C function `Execute()`.

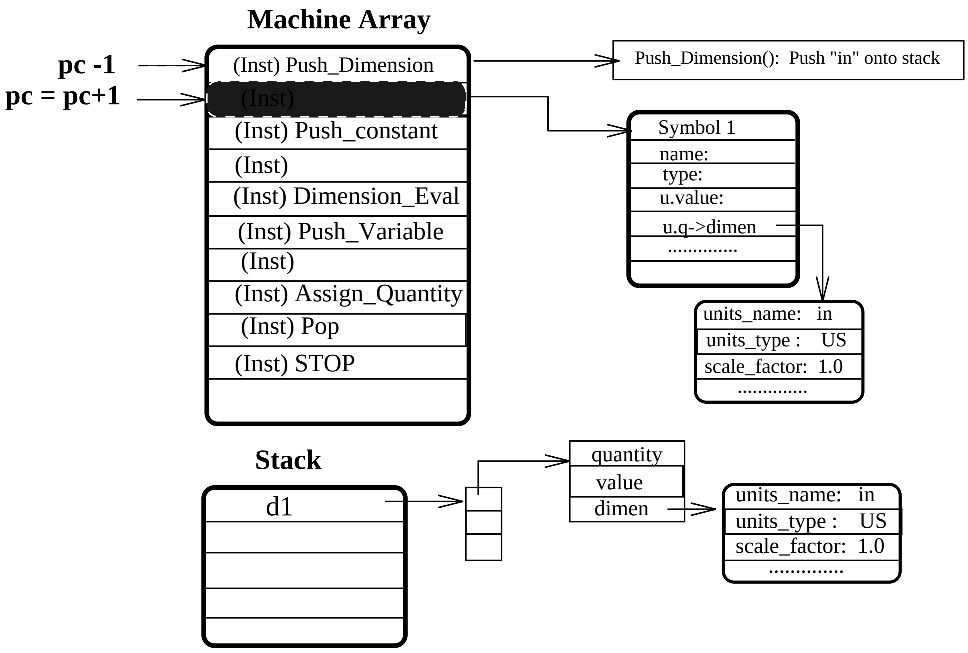


Figure 8.4: Step 1 – Push Unit onto Stack

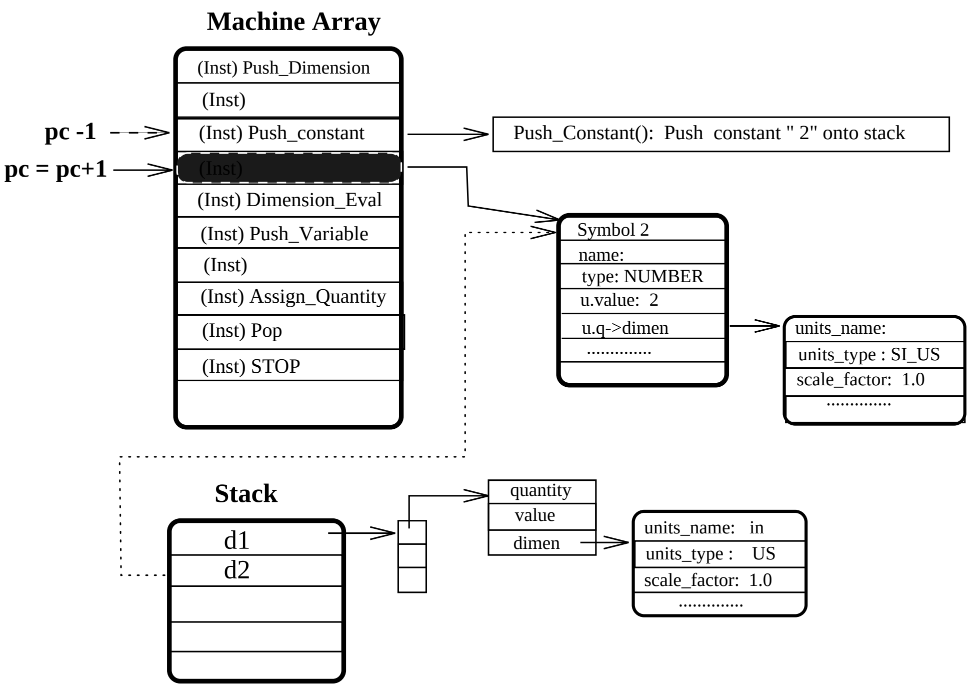


Figure 8.5: Step 2 – Push Number onto Stack

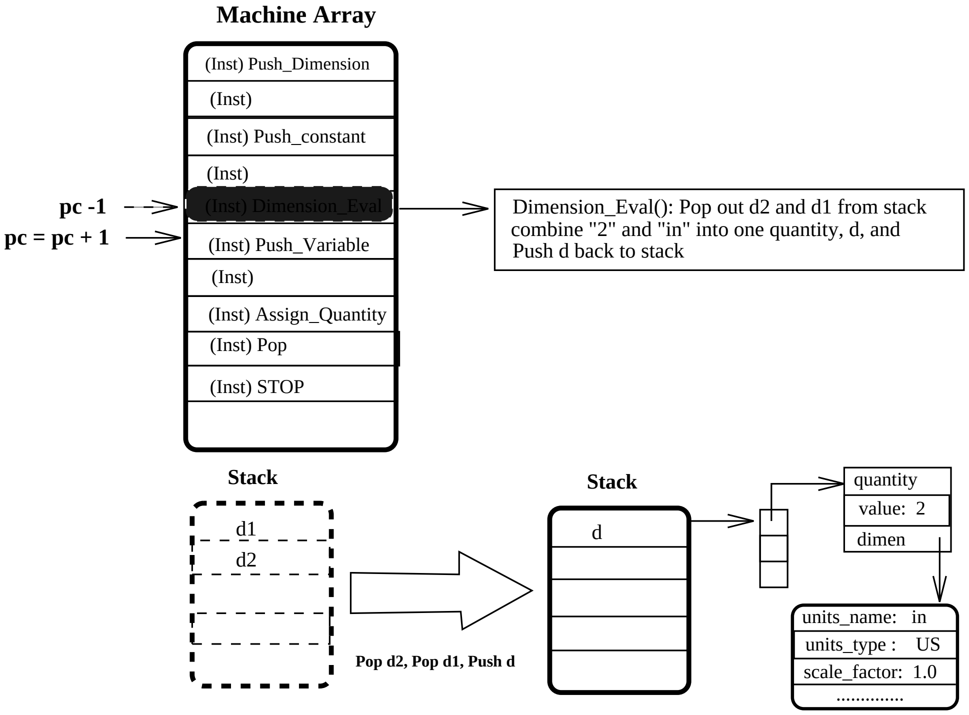


Figure 8.6: Step 3 – Combine Number and Unit into Quantity

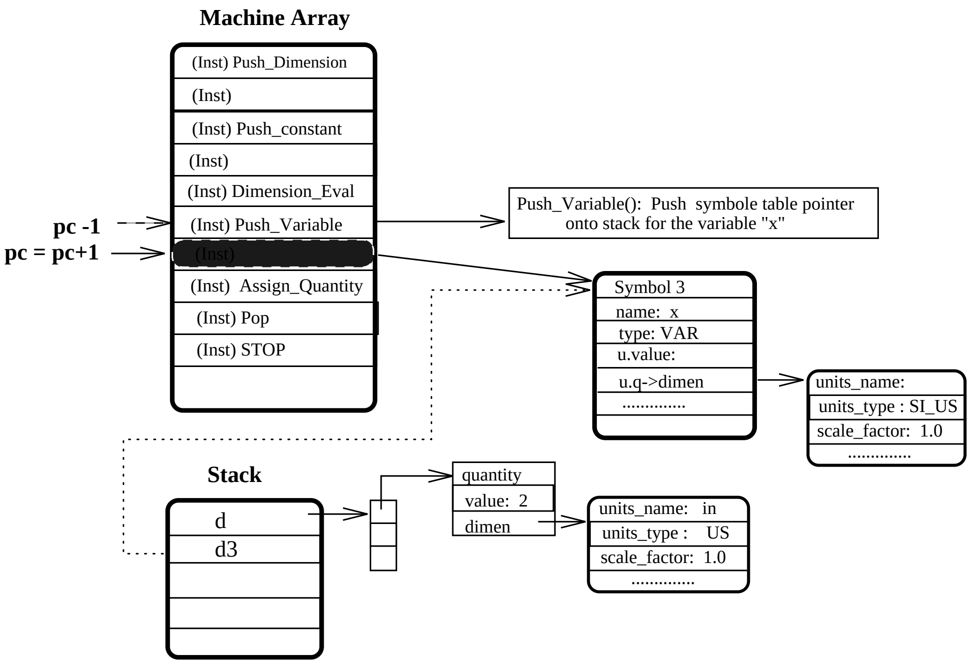


Figure 8.7: Step 4 – Push Variable onto Stack

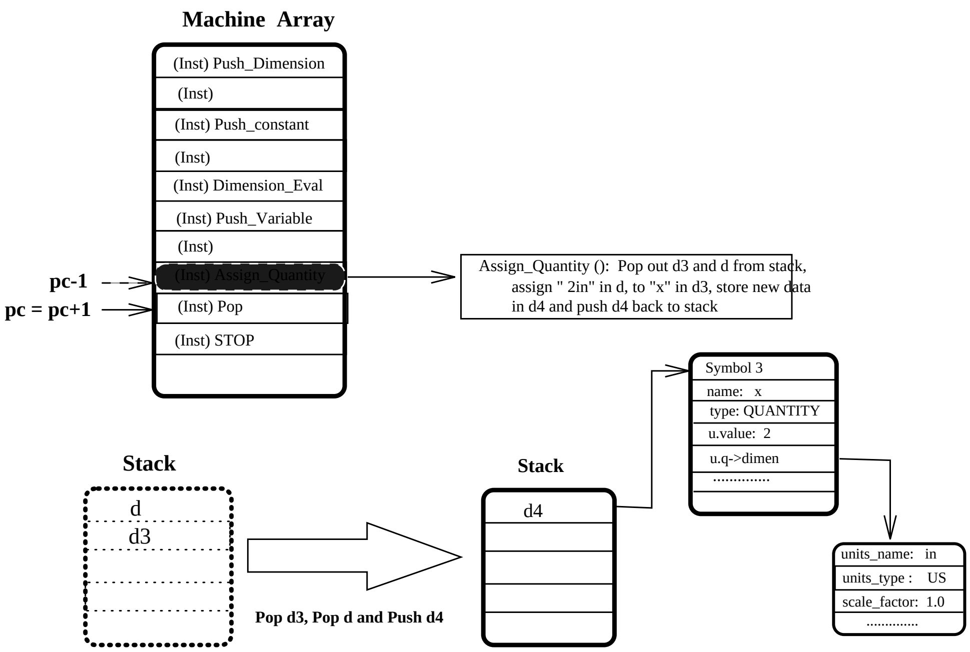


Figure 8.8: Step 5 – Assign Quantity to Variable

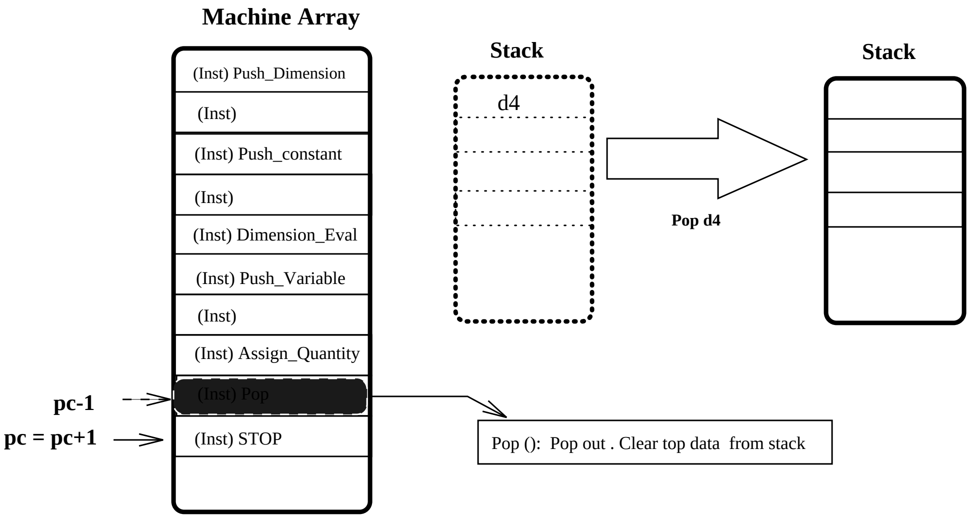


Figure 8.9: Step 6 – Clear Top Data from Stack

## 8.4 Language Design and Implementation

We have used the UNIX tool YACC (an acronym for Yet Another Compiler Compiler [16]) to define a language for the definition and solution of matrix and finite element problems. For gentle introductions to YACC, see Chapter 8 of Kernighan and Pike [18], and the O'Reilly text by Levine et al. [21]. For a theoretical discussion on LR parsers, see Aho et al. [2].

In ALADDIN, the details of the YACC grammar and associated C code are located in a file called *grammar.y*, which is partitioned into four parts:

```
%{
  Part 1 : Optional C statement, declaration;
%}
  Part 2 : YACC declarations, lexical tokens, grammar variable, precedence
           and associativity information
%%
  Part 3 : Grammar rules and semantic actions
%%
  Part 4 : Lexical analysis with a C function called yylex().
```

YACC takes the specifications in Parts 1 to 3, and generates a C code function called `yyparse()` for: (a) the matching of tokens and their types against the grammatical rules of the language, and (b) the handling of semantic actions.

`Part 4` : In Part 4 of the YACC specification, we write a C function called `yylex()` to scan streams of input characters, and identify token names and their types. The names and types of tokens are passed onto `yyparse()` for the matching against grammatical rules. Here is an abbreviated version of `yylex()`:

```
yylex()
{
double value;
int finished;

  /* [a] : Eat blank space, tab, and newline input. Read comment statements */

     while((c = fgetc(fin)) == ' ' || c == '\t' || c == '\n' || c == '/') {

          if(c == '\n')
             lineno++;

          if(fin == stdin && finp != stdin) {
             if(c == '\n') {
                printf("ALADDIN :");
                fflush(stdout);
             }
             fputc(c, finp);
          }
```

```
        if(c == '/') {
           if(follow('*', TRUE, FALSE) == TRUE) {
              c = fgetc(fin);
              finished = FALSE;
              while(finished == FALSE) {
                  c = fgetc(fin);

                  if(c == '\n')
                     lineno++;

                  if(c == EOF)
                     FatalError("ERROR >> ... file ended with unbalanced comment !!\n",
                                (char *)NULL);
                  if((c == '*') && (follow('/', TRUE, FALSE) == TRUE)) {
                     finished = TRUE;
                  }
             }
           } else {
             return c;
           }
        }
   }

/* [b] : return end-of-file */

   if(c == EOF) {
      return END_OF_FILE;
   }

/* [c] : read and store floating-point numbers */

   if (c == '.' || isdigit(c)) {
       ungetc(c, fin);
       fscanf(fin, "%lf",  &value);

       if(fin == stdin && finp != stdin) fprintf(finp, "%lf", value);

       yylval.sym = build_table("", NUMBER, value);
       return NUMBER;
   }

/* [d] : read and store variable */

   if (isalpha(c) || c == '_') {
       SYMBOL *s;
       char sbuf[100], *p = sbuf;

       do {
          if(p >= sbuf + sizeof(sbuf) - 1) {
             p = '\0';
             ExecutionError("ERROR >> name too long");
          }
          *p++ = c;
```

```
} while ((( c = fgetc(fin)) != EOF) && (isalnum(c) || (c == '_' )));

        ungetc(c, fin);
        *p = '\0';

        if(fin == stdin && finp != stdin) fprintf(finp, "%s", sbuf);

        if ((s = lookup(sbuf)) == 0) {
             s = build_table(sbuf, VAR, 0.0);
        }
        yylval.sym = s;

        switch((int) s->type) {
            case  UNDEF: case  VAR: case  QUAN:
                  return VAR;
            default:
                 return (s->type);
        }
    }

 /* [e] : Get Quoted String */

    if (c == '"') {
        char sbuf[100], *p;
        if(fin == stdin && finp != stdin) fputc(c, finp);

        for(p = sbuf; (c = fgetc(fin)) != '"'; p++) {
            if(c == '\n' || c == EOF)
               ExecutionError("ERROR >> missing quote");
            if(p >= sbuf + sizeof(sbuf) - 1) {
               p = '\0';
               ExecutionError("ERROR >> string name too long");
            }
            if((fin == stdin && finp != stdin))
              fputc(c, finp);

            *p = backslash(c);
        }
        *p = 0;

        yylval.sym = (SYMBOL *) SaveString(sbuf);

        if(fin == stdin && finp != stdin)
           fputc(c, finp);

        return STRING;
   }

   if(fin == stdin && finp != stdin) {
      fputc(c, finp);
   }

   switch(c) {
       case '>':
```

```
            return follow('=',  GE,  GT);
        case '<':
            return follow('=',  LE,  LT);
        case '=':
            return follow('=',  EQ, '=');
        case '!':
            return follow('=',  NE, NOT);
        case '|':
            return follow('|',  OR, '|');
        case '&':
            return follow('&', AND, '&');
        default:
     return c;
    }
}
```

Here we emphasize only the main components of `yylex()`. `yylex()` scans streams of input and automatically removes black spaces, tabs, and all input between comment statements. Numbers must begin with either a digit of a decimal point – they are temporarily stored in ALADDIN's symbol table. Character strings are enclosed within quotes (i.e "...."). Variables and built-in function names are alphanumeric strings that must begin with a character – the details of keywords in ALADDIN's programming language are stored in the symbol table. Finally, `yylex()` identifies token types for two-character sequences used in relational and logical operators. The interested reader should consult the program source code for details on functions `backslash()`, `follow()`, `warning()`, and so on. Now let's look at the details for our stack machine:

`Part 1` : We use the macro declarations

```
%{
...
#define   Code2(c1,c2)     Code(c1); Code(c2)
#define   Code3(c1,c2,c3)  Code(c1); Code(c2); Code(c3)
...
%}
```

as short form for calls to function `Code()`.

`Part 2` : The YACC declarations begins with character `%`.

```
%union {
     SYMBOL_PTR  sym;
     Inst      *inst;
     int        narg;
}
%token  <sym>   NUMBER
%token  <sym>   VAR
%type   <inst>  quantity
%type   <inst>  dimensions
%left    '+'   '-'              /* left  associative, same   precedence */
%left    '*'  '/'               /* left  associative, higher precedence */
%right   '^'                    /* right associative, higher precedence */
```

The `union` declares that YACC stack may holds three different types of elements; a pointer to an item in the symbol table, a machine instruction pointer, and an integer `narg` to represent the number of arguments in a function. The `token` declaration with arguements $<$ sym $>$ says that `NUMBER` and `VAR` will be input tokens accessed by dereferencing a symbol table pointer. In ALADDIN, `NUMBER` and `VAR` are integers generated by YACC to represent value of type double and character strings, respectively. The `type` declaration specifies that `quantity` and `dimensions` will be machine instructions.

The `left` and `right` keywords specify `left` and `right` associativity of operators, respectively. In YACC, the precedence of operators is defined by order of appearance, with tokens in the same declaration having equal precedence. Tokens appearing at the end of the declaration list have higher precedence than those at the beginning. Therefore, the operators `+` and `-` are left associative, and in the same precedence level. The multiply and divide operators have equal precedence, and are left associative. They have higher precedence than the add and subtract operators. The power "^" operator has the highest precedence level, and is right associative.

`Part 3 : Grammatical Rules :` ALADDIN has more than 100 grammatical rules. Rather than attempt to explain all of them here, we will concentrate only on those rules needed to match simple assignment statements, to identify matrices, and last, generate a for-looping construct. We begin with the collection of rules needed to recognize `"2"`, `"2 in"`, `"(2 in)"` and `"2 in + 3 in"`.

```
%%
quantity:  NUMBER  { $$ = Code2(Push_Dimensionless, (Inst) $1);       /* rule 1 */
                          Code2(Push_Constant, (Inst)$1);
                          Code(Dimension_Eval); }
         ;
%%
```

Number `2` is a digit. It is assigned type NUMBER in `yylex()`. A stream of input that consists of a number alone is matched by rule 1 of the grammar. A number without units is considered to be a `quantity` (i.e see left-hand side of rule 1). The name `$$` refers to value of the symbol to the left of the colon – in this case `quantity`. The semantic actions for rule 1 are a series of machine instructions, via calls to `Code()`, to push a dimensionless quantity onto the program stack, push a constant onto the program stack, and finally, evaluate the result.

We now introduce rules of the grammar designed to parse units expressions of the form $in^2$, $in * in$, $lb * in$, $in/sec^2$, and so on. The grammatical rules are:

```
dimensions: DIMENSION                { $$ = Code( Push_Dimension);   /* rule 2 */
                                       Code( (Inst)$1 );        }
          |  '(' dimensions ')'      { $$ = $2; }                    /* rule 3 */
          | dimensions '*' dimensions { Code( Dimension_Mult );  }   /* rule 4 */
```

```
        | dimensions '/' dimensions { Code( Dimension_Div);    }    /* rule 5 */
        | dimensions '^' quantity   { $$ = $1;                      /* rule 6 */
                                      Code( Dimension_Power ); }
        ;
```

YACC uses the vertical bar symbol "|" to separate individual rules in a group – the symbol should be interpreted as meaning or. Semantic actions are included inside the { }). The input streams `2 in` and `(2 in)` are considered are of type `quantity`,

```
quantity:  NUMBER dimensions { Code2(Push_Constant, (Inst)$1);
                               Code(Dimension_Eval); $$ = $2;}       /* rule 7 */
       |   '(' quantity ')'  { $$ = $2; }                            /* rule 8 */
       ;
```

and are matched by rules 7 and 8. More precisely, `(2 in)` is first matched by rule 8, and then by rule 7. The units are specified by type `dimension`.

With the rules for recognizing a quantity in place, it is relatively straight forward to define grammatical rules for arithmetic operations on quantities. In the input command `2 in + 3 in`, the quantities `2 in` and `3 in` are first recognized by rule 7, and the addition of quantities by the rule 9.

```
%%
quantity: quantity '+' quantity  { Code(Quantity_Add);  }            /* rule 9 */
%%
```

The C code located in function `Quantity_Add()` takes care of the appropriate stack operations. Similar rules are written for subtraction of two quantities (rule 10), multiplication of two quantities (rule 11), and division of quantities (rule 12). A quantity raised to the power of another quantity is considered to be a quantity (rule 13).

```
%%
quantity  : quantity '-' quantity  { Code(Quantity_Sub);  }          /* rule 10 */
          | quantity '*' quantity  { Code(Quantity_Mul);  }          /* rule 11 */
          | quantity '/' quantity  { Code(Quantity_Div);  }          /* rule 12 */
          | quantity '^' quantity  { Code(Quantity_Power);}          /* rule 13 */
          ;
%%
```

Again, `Quantity_Sub()`, `Quantity_Mul()` and so on, are C functions for arithmetic of quantities with appropriate pushing and popping of data items to/from ALADDIN's program stack. Input commands, such as `x = 3 in` and `y = 2 in + x` involve the assignment of a `quantity` to a variable. Both cases can be represented the rule:

```
%%
quantity  : VAR '=' quantity { Code2( Push_Variable, (Inst)$1);      /* rule 14 */
                               Code( Assign_Quantity); $$ = $3; }
          ;
%%
```

Using the rules stated above, `3 in` is recognized as a quantity, and `x` is a variable type. The complete input `x = 3 in` is matched by rule 14. Moreover, because YACC rules are recursively defined, `2 in + x` is matched by rule 9. Parser trees for the statements `x = 3 in` followed by `y = 2 in + x` are shown in Figure 8.10.

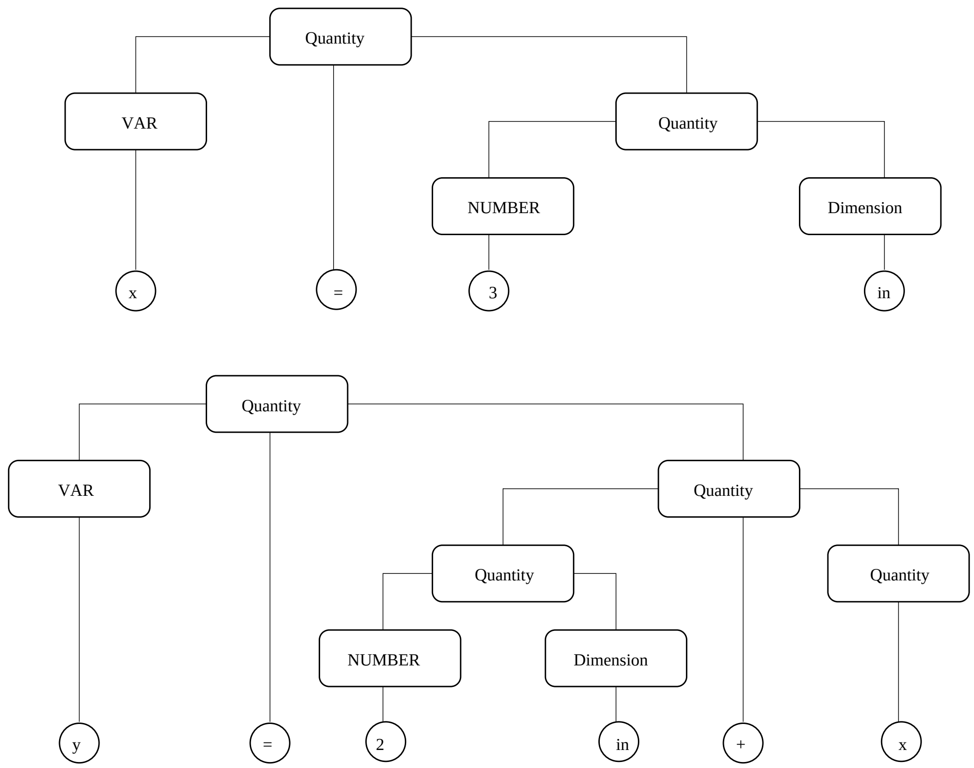


Figure 8.10: Parser Trees for `x = 3 in` and `y = 2 in + x`

**Matrix Generation :** As pointed out in the previous chapters, two mechanisms exist for the generation of matrices; definition of matrices from input, and generation of matrices via built-in matrix functions. We will begin with the latter (e.g. as in `A = Zero([2,3]);`).

```
%token  <sym>   MATX
%%
matrix : MATX      {$$ = Code( Push_Matrix );                         /* M_rule 1 */
                          Code( (Inst)$1    );
                          Code( Matrix_Eval );
%%
```

YACC uses the variable type `MATX` to represent variables of type matrix. The semantic action that follows identification of a matrix has two key parts – the C function `Push_Matrix()` pushes a pointer to the symbol table entry containing the matrix onto the program stack. `Matrix_Eval()` pops the matrix pointer from the stack, allocates memory for a new matrix, and copies the contents of the symbol table matrix to the new matrix. Finally, a pointer to the new matrix is pushed onto the stack.

The second approach is to generate matrices directly from keyboard/file input. Matrices generated directly from input are defined as series of quantities inside []. Consider, for example, the 2x2 matrix:

$$\begin{bmatrix} 2 & 3 \\ 3 & 5 \end{bmatrix}$$

which may also be represented as [ 2 , 3 ; 3, 5 ]. A matrix row is simply a list of quantities separated by commas. Individual matrix rows are separated by semicolons. The YACC grammar for identifying matrices is:

```
%type   <narg>  matrix_seq
%type   <narg>  quantity_row

%%
matrix : '[' matrix_seq ']'          {$$ = Code( Push_Variable );
                                           Code( (Inst)$2 );
                                           Code( Matrix_Build);}      /* M_rule 2 */
       ;
matrix_seq: quantity_row                 { Code( Push_Variable );
                                           Code( (Inst)$1 );
                                           $$ = 1; }                  /* M_rule 3 */
          | matrix_seq ';' quantity_row  { Code2( Push_Variable );
                                           Code( (Inst)$3 );
                                           $$ = $1 + 1;}              /* M_rule 4 */
       ;
```

where `type` specifies that `matrix_seq` and `quantity_row` are of type $<$ narg $>$. `matrix_seq` records number of rows and the number is pushed onto stack – this is done by `M_rule 3` and `M_rule 4`. As soon as row of quantities is identified, `matrix_seq` is initialized with number = 1 (`M_rule 2`). Then, `matrix_seq` is increased by 1 for each semicolon ";" that is parsed, with the sum being accumulated recursively via `M_rule 4`. Within each row, columns separated by commas (",") are quantities.

```
quantity_row:                            { $$ = 0; }
       | quantity                        { $$ = 1; }                  /* QR_rule 1 */
       | quantity_row ',' quantity       { $$ = $1 + 1;}              /* QR_rule 2 */
       ;
```

Similarly, the number of columns in a matrix is recorded by `quantity_row`.

**Matrix Operations :** The grammatical rules for matrix operations are similar to those of engineering quantities.

```
  | matrix '*' matrix   {Code( Bltin_Matrix_Mult);}                   /* M_rule 5  */
  | matrix '+' matrix   {Code( Bltin_Matrix_Add);}                    /* M_rule 6  */
  | matrix '-' matrix   {Code( Bltin_Matrix_Sub);}                    /* M_rule 7  */
  | '(' matrix ')'      {$$ = $2;}                                    /* M_rule 8  */
  | quantity '*' matrix {Code( Bltin_Quan_Matrix_Mult);
```

```
                        $$ = $3;}                                   /* M_rule 9  */
  | matrix '*' quantity {Code( Bltin_Matrix_Quan_Mult);}             /* M_rule 10 */
  | matrix '/' quantity {Code( Bltin_Matrix_Quan_Div);}              /* M_rule 11 */
  ;
```

These rules generate an array of machine instructions for matrix addition, subtraction and multiplication (i.e. `M_rule 5`, `M_rule 6` and `M_rule 7`). They allow for the parsing of a matrix inside parentheses (`M_rule 8`), the multiplication of a quantity times a matrix (`M_rule 9`), a matrix times a quantity (`M_rule 10`), and a matrix divided by a quantity (`M_rule 11`).

**For Looping Construct :** The development of a YACC grammar and supporting C code for looping and branching constructs is the most complicated part of ALADDIN's programming language. A description of the while-looping construct may be found in Chapter 8 of Kernighan and Pike [18]. The purpose of this subsection is to explain our implementation of the for-looping construct. The family of grammatical rules:

```
stmt: for '(' exprlist ';' cond ';' exprlist ')' stmt end {      /* F_rule 0 */
        ($1)[1] = (Inst) $3;              /* initiation          F_rule 1 */
        ($1)[2] = (Inst) $5;              /* condition           F_rule 2 */
        ($1)[3] = (Inst) $9;              /* body of loop        F_rule 3 */
        ($1)[4] = (Inst) $7;              /* increments          F_rule 4 */
        ($1)[5] = (Inst) $10;             /* end, if cond fails  F_rule 5 */
        }
      ;

for:  FOR { $$ = Code3( For_Code, STOP, STOP); Code2(STOP, STOP); Code(STOP);}
   ;

exprlist: listlist { Code(STOP); $$ = $1; }
          ;
listlist :     { $$ = progp; }                                 /* L_rule 1 */
         | quantity                                            /* L_rule 2 */
         | listlist ',' quantity                               /* L_rule 3 */
         ;
cond:  quantity { Code(STOP); $$ = $1; }
    ;
```

assumes that memory in the array of machine instructions will take a form as shown in Figure 8.11. A for-loop is simply a statement (`stmt`). The initiation and incremental steps of the for-loop are expressed as `exprlists`, which are themselves subdivided into `listlist`. A `listlist` could be a space (nothing) (`L_rule 1`), an engineering quantity (`L_rule 2`), or a series of quantities separated with "," (`L_rule 3`). The body of a for-loop is also a statement(`stmt`) – this rule allows for the nesting of for-loop constructs. The condition part of the for loop is an engineering quantity.

After the token `FOR` has been recognized, five spaces in the array of machine instructions are reserved for the storage of pointers to the initiation code (`F_rule 1`), the

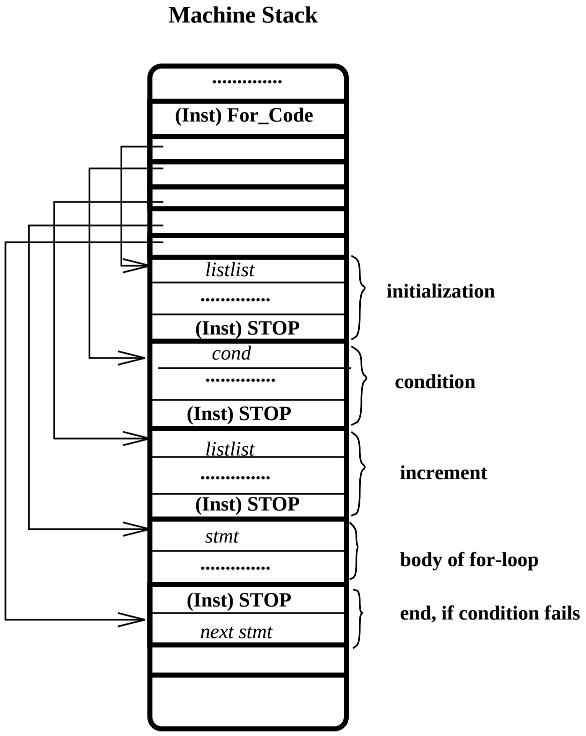


Figure 8.11: Machine Stack for For-Loop Construct

condition (`F_rule 2`), the incremental steps (`F_rule 3`), the body of the for-loop (`F_rule 4`) and the `STOP` instruction (`F_rule 5`). The two character sequence refers `$n` refers to the value of a type stacked n spaces beyond the current namelist production. In this particular case, `$1` is the location in the machine of the token identified by `F_rule 0`. The initiation, condition, body, increments, and end components of the for loop are access by machine array elements `($1)[1]` to `($1)[5]`, respectively. For the sake of completeness, we now include the C code for `For_Code()`.

```
For_Code()
{
DATUM d;
Inst *savepc = pc;

     Execute(*((Inst **) savepc)) ;     /* initiation */
     Execute(*((Inst **) (savepc+1))) ; /* cond */

     d = Pop();
     while(d.q->value) {
        if(Check_Break()) break;

        Execute(*((Inst **)(savepc+2))); /* body of loop */
        Execute(*((Inst **)(savepc+3))); /* increments */
        if(returning || Check_Break()) break;
        Execute(*((Inst **) (savepc+1))) ; /* cond */

        if( CheckUnits()== ON ) {
```

```
            free((char *)d.q->dimen->units_name);
            free((char *)d.q->dimen);
            free((char *)d.q);
        }
        d = Pop();
    }

    if(!returning)
        pc = *((Inst **)(savepc + 4)); /* next statement */
    After_Break();
}
```

Notice how the layout of memory in Figure 8.11 has been carefully combined with function calls to `Execute()`.

# Part V

# CONCLUSIONS AND FUTURE WORK

# Chapter 9

# Conclusions and Future Work

## 9.1 Conclusions

In the general area of computational technology for engineering, the 1990's are shaping up to be the decade of global networking and multimedia. These advances will allow for the prototyping of interactive computing environments tailored to the life cycle development of complex engineering systems. For these environments to be commercially successful, they will need to support development of engineering systems from multiple viewpoints, and at multiple levels of abstraction. Engineers should be provided with the tools to integrate once disparate disparate disciplines. While the merging of matrix computations with finite elements, optimization, and graphics is already evident, many other possibilities exist. For instance, now that many companies are engaging in business practices that are geographically dispersed, engineers will soon expect connectivity to a wide range of information services – electronic contract negotations, management of project requirements, design rule checking over the Internet, and availability of materials and construction services are likely candidates. The participating computer programs and networking infrastructure will need to be fast and accurate, flexible, reliable, and of course, easy to use [12, 28].

Our long-term goal for ALADDIN is to provide engineers with such a computational problem-solving environment. Version 1.0 of ALADDIN is simply a first step in this direction, and is much less ambitious. The emphasis in development has so far been on the basic system specification, and its application to matrix and finite element problems. We have tried to determine what data types, control structures, and functions will allow engineers to solve a wide variety of problems, without the language becoming too large and complicated, and without extensibility of the system being compromised. Our strategy for achieving the latter objective is described in Chapter 1. The test applications have been taken from traditional Civil Engineering, and have focussed on the design and analysis of highway bridges and earthquake-resistant buildings.

ALADDIN (Version 1.0) is approximately 36,000 lines of C, and getting the program to this point has taken approximately three years of part-time work. We have written nearly all of the source code from scratch – along the way, a new data structure for the matrices with units was designed and software implemented. Many of the matrix algorithms that ALADDIN uses are documented in the forthcoming text of Austin and Mazzoni [5]. The language design must be followed by entensive testing of the software with matrix and finite element test problems that are complicated enough to push the limits of the software (Note : In our experience, many coding errors will not show up in trivial engineering applications).

As the work on ALADDIN progressed, we occasionally identified small extensions to the language that would allow ALADDIN to attack a new problem area. When the language for ALADDIN was first designed we did not have design rule checking in mind, for example. Yet once the basic matrix and function language specification was in place, and the finite element analyses worked, only a couple of extra functions were needed for the implementation of basic design rule checking.

Together these observations lead us to believe that our goal of keeping the ALADDIN language small and extensible has been achieved. This work lays the foundation for many avenues of future work.

## 9.2 Future Work

Figure 9.1 is a schematic of the current components of work, plus modules of work that are needed for the next extension of ALADDIN. The future work should focus on:

[1] `Nonlinear Algorithms` : The scope of applications in this report has been restricted to time-dependent analysis of linear structural systems. Future work should extend this work to the time-history analysis of systems having material and geometric nonlinearities. A first step in this direction is reported in Chen and Austin [10].

[2] `User-defined Functions` : Version 1.0 of ALADDIN does not permit the use of user-defined functions and procedures in the input file. When we first began working on ALADDIN, our feeling was that the input file language should be relatively simple – otherwise, why not simply code the whole problem in C ? An easy way of controling language complexity is to disallow user-defined functions in the input file, and this is what we have done for ALADDIN Version 1.0. Now that the matrix and finite element libraries are working, this early decision needs to be revised. We would like to combine the finite element analysis with discrete simulated annealing optimization algorithms, along the lines proposed by Kirkpatrick [19]. The latter could be succinctly implemented if user-defined functions were available. The current version of ALADDIN assumes that all user-defined variables are global. When user-defined functions are added to ALADDIN, notions of scope should also be added to variables (this could be implemented via an array of Symbol Tables).

[3] `Graphical Library` : An obvious limitation of ALADDIN (Version 1.0) is that it lacks a graphical user interface. Providing a suitable interface is easier said than

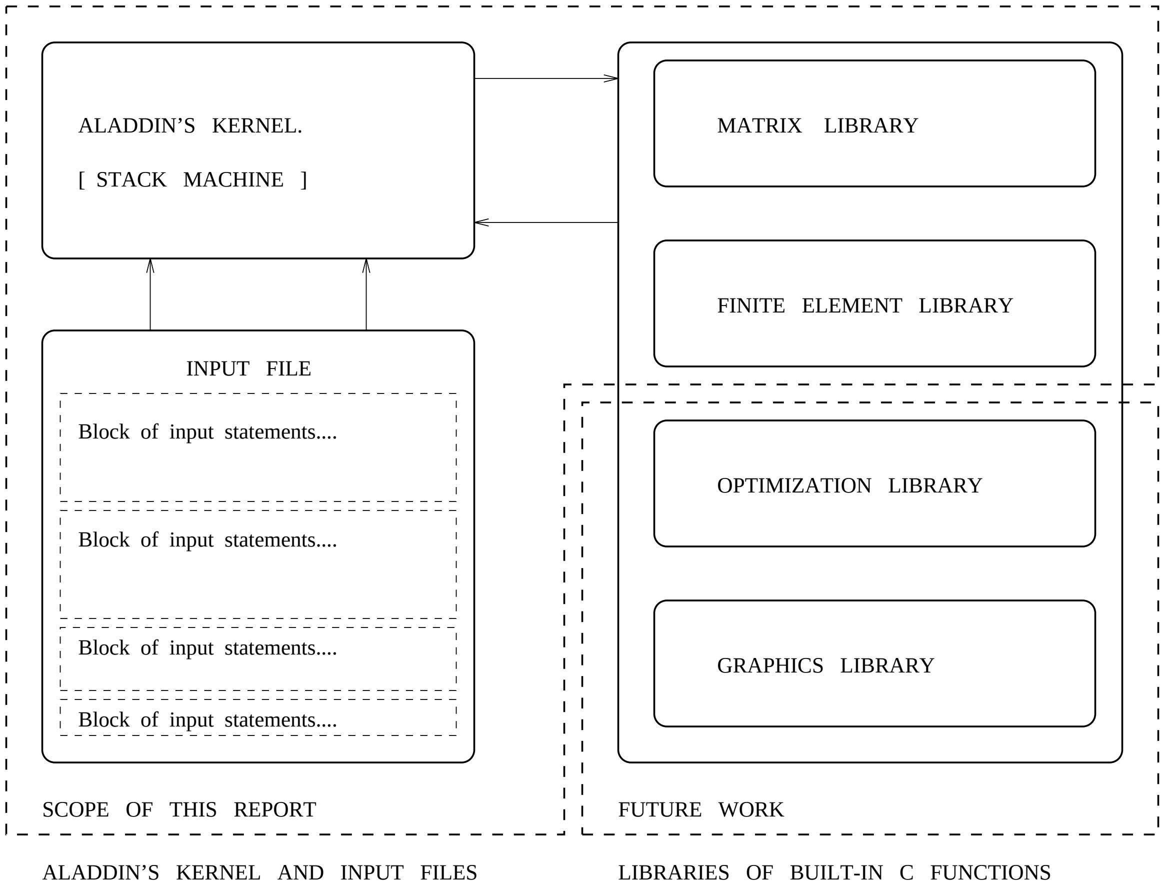


Figure 9.1: Future Development for ALADDIN

done. If ALADDIN simply preported to conduct finite element analysis, then it would be a relatively easy process to code-up an interactive pre- and post-finite element processor in X windows (or a suitable PC graphics package).

The problem with this approach is that it only serves one half of ALADDIN's capabilities – in addition to graphically describing the finite element mesh and modeling assumptions, a graphical interface should also provide a graphical counterpart to the solution algorithm specifications (e.g. Newmark Integration). We will explore the use of visual programming languages to see if they are suitable for this purpose.

[4] `Optimization Library` : We would like to extend the language specification so that engineers can describe design objectives, design constraints, and design parameters, for general engineering optimization problems in a compact manner. The issue of "compactness" is particularly relevant for optimization problems that involve finite element analysis because they often contain hundreds, and sometimes thousands, of constraints.

Further work is needed to determine how groups of similar constraints can be bundled into groups, and how ALADDIN's control structures can be exploited to write down these constraints in a compact manner. Once the issue of constraints is sorted out, the pathway for describing the design parameters and objectives should (hopefully) become clearer.

[5] `Test Problem Suite` : New suites of test problem files will be needed as the details of items [2] and [4] are filled in (please let us know if you find an error in one of the test files!).

Version 1.0 of ALADDIN is basic in the sense that many features found in commercial finite element codes have yet to be implemented in ALADDIN. For example, the current version of ALADDIN only supports external loads applied at the nodes. In the near future we will add element level loads so that uniform loads can be applied to two-dimensional problems, and pressure loads can be applied to the surface of shell finite elements in three-dimensional problems.

[6] `WWW Home Page` : We are currently developing a World Wide Web (WWW) Home Page for ALADDIN. The ALADDIN home page will provide basic information on ALADDIN, sample matrix and finite element test problems, ALADDIN source code, and copies of this report. Our near-term plans are to experiment with the dynamic documents capabilities of WWW, and create home pages where WWW visitors can interact with an ALADDIN script running at the University of Maryland.

Work items [1] to [5] are tightly coupled. Understanding their interaction is an active area of research.

**Acknowledgements**

This work has been supported in part by NSF Grant ECD-8803012, by NSF grants to the Institute for Systems Research at the University of Maryland, and by a series of fellowship grants to Wane-Jang Lin from the Federal Highway Administration.

The early stages of this project would not have been possible without valuable discussions with Dr. Bunu Alibe and Bert Davy, faculty members in the Department of Civil Engineering at Morgan State University.

# Bibliography


[1] ABAQUS. *Example Problems Manual, version 5.2.* Hibbitt, Karlsson & Sorensen, Inc., Hibbitt, Karlsson & Sorensen, Inc., 1080 Main Str., Pawtuchet, RI 02860, 1992.

[2] Aho A. V., Sethi R., Ullman J. D. *Compilers : Principles, Techniques and Tools.* Addison Wesley, 1985.

[3] Austin M. A., Pister K. S., Mahin S. A. A Methodology for the Probabilistic Limit States Design of Earthquake Resistant Structures. *Journal of the Structural Division, ASCE*, 113(8):1642–1659, August 1987.

[4] Austin M. A., Pister K. S., Mahin S. A. Probabilistic Limit States Design of Moment Resistant Frames Under Seismic Loading. *Journal of the Structural Division, ASCE*, 113(8):1660–1677, August 1987.

[5] Austin M.A., Mazzoni D. *C Programming for Engineers.* John-Wiley and Sons. (Scheduled for publication in 1996).

[6] Balling R. J., Pister K. S., Polak E. DELIGHT.STRUCT: A Computer-Aided Design Environment for Structural Engineering. *Computer Methods in Applied Mechanics and Engineering*, pages 237–251, January 1983.

[7] Bathe K. J. *Numerical Methods in Finite Element Analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

[8] Bathe K. J. *Finite Element Procedures in Engineering Analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1982.

[9] Bathe K. J., Cimento, A. P. Some Practical Procedures for the Solution of Nonlinear Finite Element Equations. *Computer Methods in Applied Mechanics and Engineering*, 22:59–85, 1980.

[10] Chen X.G., Austin M.A. Development of a Shell Finite Element for Project ALADDIN. Technical report, University of Maryland, College Park, MD. 20742, 1995.

[11] Cmelik R.F., Gehani N.H. Dimensional Analysis with C++. *IEEE Software*, pages 21–26, May 1988.

[12] Gallopoulos E., Houstis E. Computer as a Thinker/Doer : Problem-Solving Enviroments for Computational Science. *IEEE Computational Science and Engineering*, 1(2):11–23, 1994.

[13] Gehani N.H. Databases and Units of Measure. *IEEE Transactions on Software Engineering*, 8(6):605–611, November 1982.

[14] Ghali A., Neville A.M. *Structural Analysis : A Unified and Matrix Approach.* Chapman and Hall, 2nd Edition edition, 1979.

[15] Jin Lanheng. Analysis and Evaluation of a Shell Finite Element with Drilling Degree of Freedom. Technical report, University of Maryland, College Park, Maryland 20742, 1994.

[16] Johnson S. C. YACC - Yet another Compiler Compiler. Technical report, AT&T Bell Laboratories, Murray Hill, New Jersey, 1975.

[17] Karr M., Loveman D. B. Incorporating Units into Programming Languages. *Communications of the ACM*, 21(5):385–391, May 1978.

[18] Kernighan B. W., Pike R. *The UNIX Programming Environment.* Prentice-Hall Software Series, 1983.

[19] Kirkpatrick S., Gelatt C.D., Vecchi M.P,. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

[20] Kronlof K. *Method Integration : Concepts and Case Studies.* John-Wiley and Sons, 1993.

[21] Levine J.R., Mason T., Brown D. *Lex and Yacc.* O'Reilly and Associates, Inc, 1992.

[22] Luenberger David G. *Linear and Nonlinear Programming.* Addison-Wesley Publishing Company, 1984.

[23] Mondkar D. P., Powell G. H. ANSR - 1 General Purpose program for Analysis of Nonlinear Structural Response. *Report No.* EERC 75-37, Earthquake Engineering Research Center, December 1975.

[24] Nye W. T., Riley D. C., Sangiovanni-Vincentelli A. L., Tits A. L. DELIGHT.SPICE: An Optimization-Based System for the Design of Integrated Circuits. *IEEE Trans. CAD Integrated Circuits and Systems, CAD-7*, pages 501–520, 1987.

[25] Nye W. T., Tits A. L. An Application-Oriented, Optimization-Based Methodology for Interactive Design of Engineering Systems. *International Journal of Control*, 43(6):1693–1721, 1986.

[26] Nye W., Tits A. L. DELIGHT FOR BEGINNERS. Memorandum No. UCB/ERL M82/55, Department of Electrical Engineering, July 1982.

[27] Salter K.G. A Methodology for Decomposing System Requirements into Data Processing Requirements. *Proc. 2nd Int. Conf. on Software Engineering*, 1976.

[28] Tesler L.G. Networked Computing in the 1990's. *Scientific American*, 265(3):86–93, September 1991.

[29] Wu T.L. DELIGHT.MIMO : An Interactive System for Optimization-Based Multivariable Control System Design. Memorandum No. UCB/ERL M86/90, Department of Electrical Engineering, December 1986.

[30] Zienkiewitz O.C., Taylor R.L. *The Finite Element Method.* McGraw-Hill, Inc, 1989. For details on FEAP, see Chapter 15.