IMPLEMENTATION OF ONE AND TWO DIMENSIONAL LINEAR FINITE ELEMENT PROGRAMS WITH JAVA LANGUAGE

by

HİLMİ AYDIN İŞBAŞAR

B.S., Civil Engineering, İstanbul Technical University, 2001

Submitted to the Institute for Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science

Graduate Program in Civil Engineering
Boğaziçi University

2008

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to my supervisor Prof. Cengiz Karakoç for all the guidance, moral contribution and encouragement he has offered me throughout the period of this thesis.

I would also like to express my special thanks to Assoc. Prof. Orhun Köksal and Assist. Prof. Kutay Orakçal for their discussions and making helpful comments.

Finally, I offer my sincere gratitude to my family for their support, encouragement and patience.

ABSTRACT

IMPLEMENTATION OF ONE AND TWO DIMENSIONAL LINEAR FINITE ELEMENT PROGRAMS WITH JAVA LANGUAGE

Finite element program codes were generally developed in Fortran but during last decade developers started to seek better programming paradigms in order to handle complexity in finite element software. Since Java language developed by Sun Microsystems possesses features, which makes it attractive for using in finite element modeling, in this study Java programming language is chosen to develop a well organized finite element program.

Following the discussion of the finite element method, one dimensional rod and beam elements are implemented to 3D Truss and Beam-Column Frame programs respectively. Moreover, by implementing two dimensional high order quadrilateral and triangular elements the Plane Stress program is developed. Performances of the implemented Java programs are compared with the commercial software packages and exact solutions.

In order to illustrate the comparison of the results, five different case studies are investigated. 3DTruss and Beam-Column Frame programs are compared with SAP2000 software. Plane Stress-Strain program is firstly compared with LUSAS software and secondly with ANSYS software.

Results of this comparison revealed the efficiency and accuracy of the Java finite element programs implemented in this study.

ÖZET

BİR VE İKİ BOYUTLU LİNEER SONLU ELEMAN PROGRAMLARININ JAVA DİLİ İLE UYGULAMASI

Sonlu eleman program kodları gelenekesel olarak Fortran dilininde geliştirilmiştir fakat son yıllarda karmaşık yapılı sonlu eleman programlarının geliştirilmesinde daha gelişmiş programlama altyapıları için arayışlar başlamıştır. Sun Microsystem tarafından geliştirilen Java programlama dili, bünyesinde sonlu elelemanlar modellemeleri için dikkat çekici özellikler barındırdığı için bu çalışmada, Java programlama dili seçilerek yeni bir sonlu elemanlar programı geliştirilmiştir.

Sonlu elemanlar methodunun anlatımının ardından, bir boyutlu sonlu elemanlar kullanılarak uzay kafes ve çerçeve programları geliştirilmiştir. Sonrasında, iki boyutlu yüksek mertebeli izoparametrik elemenlar kullanılarak düzlem gerilme programı geliştirilmiştir. Geliştirilen Java programları, ticari paket programlarla ve gerçek sonuçlarla karşılaştırılmıştır.

Değerlendirmelerin yapılabilmesi için, 5 farklı problem incelenmiştir. Uzay kafes ve çerçeve programları SAP2000 programı ile karşılaştırımıştır. Düzlem gerilme programı ilk olarak LUSAS ardından ANSYS paket programları ile karşılaştırılmıştır.

Karşılaştırmanın sonuçları, bu çalışmada Java dili ile yazılan sonlu eleman programlarının doğruluğunu ve kesinliğini teyit etmektedir.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT iv
ÖZET v
LIST OF FIGURES xi
LIST OF TABLES xvi
LIST OF SYMBOLS/ABBREVIATIONS xviii
1. INTRODUCTION TO THE FINITE ELEMENT METHOD AND THE JAVA LANGUAGE 1
1.1. Historical Background of the Finite Element Method 1
1.2. Computer Science Revolution 3
1.3. The Birth of the Java Language 4
1.4. The Advantages of the Java Language 4
1.4.1. Platform Independency 4
1.4.2. Object Oriented Programming Paradigm 5
1.4.3. Simplicity of Programming 6
1.4.4. Built-in Multithreading 6
1.4.5. Flexibility and Presentation 7
1.5. Performance of the Java Language 7
2. FUNDAMENTALS OF THE FINITE ELEMENT METHOD 9
2.1. Mathematical Theory 9
2.1.1. Galerkin Method of Weighted Residuals 9
2.2. Derivation of One Dimensional Element Stiffness 13
2.2.1. The Rod Element Stiffness 13
2.2.2. The Beam Element Stiffness 15
2.2.3. Derivation of One Dimensional Element Stiffness by Energy Approach 18
2.3. Master Stiffness Method 20
2.3.1. Introduction 20
2.3.2. Element Stiffness Matrix in Local Coordinates 21
2.3.3. Element Stiffness Matrix in Global Coordinates 22
2.3.4. Assembly of Global Structural Stiffness Matrix 25
2.3.5. Reduced Global Structural Stiffness System Calculations 28

2.3.6. Solution for the Unknown Displacements 29
2.3.7. Calculations of Reaction Forces 30
2.3.8. Calculations of Internal Forces and Stresses 31
2.4. Derivation of Two Dimensional Element Stiffness 33
2.4.1. Derivation of Two Dimensional Element Stiffness by Energy Approach 35
3. JAVA IMPLEMENTATION OF MASTER STIFFNESS METHOD 37
3.1. Implementation of Global Element Stiffness Matrix 37
3.1.1. Formal Parameters of Rod2Stiff 38
3.1.2. Example Commands for Rod2Stiff 39
3.2. Implementation of Global Structural Stiffness Matrix 40
3.2.1. Formal Parameters of PTGlobStrStiff 40
3.2.2. Example Command for PTGlobStrStiff 42
3.3. Implementation of Reduced Global Structural Stiffness System 44
3.3.1. Formal Parameters of ReducedGlobStrStiff 44
3.3.2. Example Command for ReducedGlobStrStiff 46
3.4. Implementation of External Loads Considering Prescribed Displacements . . 46
3.4.1. Formal Parameters of ReducedNodeForces 47
3.4.2. Example Command for ReducedNodeForces 49
3.5. Implementation of Displacement Solutions 49
3.5.1. Formal Parameters of JAMA.Matrix 50
3.5.2. Example Command for JAMA.Matrix 50
3.6. Implementation of Node Reaction Force Solutions 51
3.6.1. Example Calculation of Node Reaction Forces 51
3.7. Implementation of Internal Force and Stress Solutions 52
3.7.1. Formal Parameters of Rod2InternalFrc 52
3.7.2. Formal Parameters of Rod2Strs 53
4. JAVA PROGRAM FOR 3D TRUSSES 54
4.1. Introduction 54
4.2. Implementation of 3D Rod Element 55
4.2.1. Formal Parameters of a3DRod3Stiff 56
4.3. Implementation of Global Structural Stiffness Matrix 56
4.3.1. Formal Parameters of a3DTrussGlobStrStiff 58
4.4. Implementation of Reduced Global Structural Stiffness System 58

4.5. Implementation of External Loads Considering Prescribed Displacements . . 59
4.6. Implementation of Displacement Solutions 59
4.6.1. Formal Parameters of a3DTrussSolution 60
4.7. Implementation of Internal Force and Stress Solutions 61
4.7.1. Formal Parameters of a3DTrussInternalFrc 61
4.7.2. Formal Parameters of a3DTrussStrs 63
4.8. Implementation of Results 63
4.9. Flow Chart of the 3D Truss Program 64
4.10.Case Study of a Plane Truss Problem 66
4.11.Case Study of a 3D Truss Problem 68
5. JAVA PROGRAM FOR BEAM-COLUMN FRAME STRUCTURES 72
5.1. Introduction 72
5.2. Implementation of Beam Element Stiffness Matrix 73
5.2.1. Formal Parameters of Beam3Stiff 75
5.3. Implementation of Global Structural Stiffness Matrix 76
5.3.1. Formal Parameters of PBGlobStrStiff 77
5.4. Implementation of Reduced Global Structural Stiffness System 77
5.5. Implementation of External Loads 78
5.5.1. Formal Parameters of PBReducedNodeForces 78
5.6. Implementation of Displacement Solutions 79
5.6.1. Formal Parameters of PBSolution 79
5.7. Implementation of Axial Forces and Bending Moments 80
5.7.1. Formal Parameters of BPIntForces 80
5.8. Implementation of Results 82
5.9. Flow Chart of the Beam-Column Frame Program 82
5.10.Case Study of a Beam-Column Frame Problem 84
6. JAVA IMPLEMANTATION OF QUADRILATERAL ELEMENTS 88
6.1. Introduction 88
6.2. Computation of Strain-Displacement Matrix 91
6.2.1. The Jacobian Transformation 91
6.2.2. Shape Function Derivatives 92
6.2.3. Calculating Jacobian Matrix 92
6.2.4. Derivation of Stiffness Matrix 93

6.3. Implementation of Gauss Rule for Quadrilateral Elements 94
6.3.1. Formal Parameters of Gauss1D. 95
6.3.2. Example Commands for Gauss1D 96
6.3.3. Formal Parameters of Gauss2D. 98
6.3.4. Example Commands for Gauss2D 98
6.4. Implementation of Stiffness Matrix of 4 Point Quadrilateral 100
6.4.1. Formal Parameters of Pnt4QdrMembStiff 100
6.4.2. Example Commands for Pnt4QdrMembStiff 103
6.5. Implementation of Stiffness Matrix of 8 Point Quadrilateral 104
6.5.1. Formal Parameters of Pnt8QdrMembStiff 105
6.6. Implementation of Stiffness Matrix of 9 Point Quadrilateral 108
6.6.1. Formal Parameters of Pnt9QdrMembStiff 108
7. JAVA IMPLEMANTATION OF TRIANGULAR ELEMENTS. 111
7.1. Introduction 111
7.2. Implementation of Stiffness Matrix of Constant Stress Triangle 113
7.2.1. Formal Parameters of Pnt3TrgMembStiff 114
7.3. Implementation of Gauss Rule for Triangular Elements 115
7.3.1. Formal Parameters of TriangularGauss 117
7.3.2. Example Command for TriangularGauss 118
7.4. Implementation of Stiffness Matrix of 6 Point Isoparametric Triangle 118
7.4.1. Formal Parameters of Pnt6TrgMembStiff 119
8. JAVA PROGRAM FOR PLANE STRESS PROBLEMS 123
8.1. Introduction 123
8.2. Implementation of Global Structural Stiffness Matrix 123
8.2.1. Formal Parameters of PSGlobStrStiff 124
8.3. Implementation of Reduced Global Structural Stiffness System 126
8.4. Implementation of External Loads 126
8.5. Implementation of Displacement Solutions. 127
8.5.1. Formal Parameters of PSSolution 128
8.6. Implementation of Stresses of Two Dimensional Elements 128
8.6.1. Introduction 128
8.6.2. Formal Parameters of Pnt4QdrMembStresses 129
8.6.3. Formal Parameters of Pnt8QdrMembStresses 130

8.6.4. Formal Parameters of Pnt9QdrMembStresses 131
8.6.5. Formal Parameters of Pnt3TrgMembStresses 132
8.6.6. Formal Parameters of Pnt6TrgMembStresses 132
8.7. Implementation of Stress Smoothing Procedure 133
8.7.1. Introduction 133
8.7.2. Formal Parameters of StressAveraging 134
8.8. Implementation of Results 137
8.9. Flow Chart of the Plane Stress Program 138
9. PLANE STRESS CASE OF A CANTILEVER BEAM 139
9.1. Introduction 139
9.2. Determination of the Exact Solutions 140
9.3. Data Entry 141
9.4. Comparison of the Results with LUSAS Software 144
10.PLANE STRESS CASE OF A SIMPLY SUPPORTED BEAM 167
10.1. Introduction 167
10.2. Determination of the Exact Solutions 168
10.3. Data Entry 169
10.4. Comparison of the Results with ANSYS Software 170
11. CONCLUSIONS 177
APPENDIX A : CONVERGENCE OF ANALYTIC RESULTS 179
A.1. Consistency 179
A.2. Stability 180
REFERENCES 182
REFERENCES NOT CITED 184
APPENDIX B : JAVA PROGRAM 186

LIST OF FIGURES

Figure 1.1. Finite element meshing 2

Figure 2.1. Finite element representation of u(x) 10

Figure 2.2. Sample plane truss 13

Figure 2.3. Rod element 13

Figure 2.4. Rod element equilibrium 14

Figure 2.5. Beam element 15

Figure 2.6. End displacements and end forces 21

Figure 2.7. Node displacement transformations 22

Figure 2.8. Node force transformations 23

Figure 2.9. Sample structure 26

Figure 2.10. Sample element 1. 26

Figure 2.11. Sample element 2. 27

Figure 2.12. Sample external load. 29

Figure 2.13. Reaction forces of the sample structure 31

Figure 2.14. Internal forces of the sample structure 31

Figure 3.1. Plane truss examples 37

Figure 3.2. Rod element in local coordinates 38

Figure 3.3. Rod2Stiff module. 39

Figure 3.4. Rod2Stiff example 39

Figure 3.5. PTGlobStrStiff module 42

Figure 3.6. PTGlobStrStiff example 43

Figure 3.7. ReducedGlobStrStiff module 45

Figure 3.8. Initial settlement sample 47

Figure 3.9. ReducedNodeForces module 48
Figure 3.10. Jama matrix module 50
Figure 3.11. Rod2InternatFrc module 52
Figure 3.12. Rod2Strs module 53
Figure 4.1. 3D Truss example 54
Figure 4.2. 3D Rod element. 55
Figure 4.3. a3DRod3Stiff module 56
Figure 4.4. a3DTrussGlobStrStiff module 57
Figure 4.5. a3DTrussSolution module 60
Figure 4.6. a3DTrussInternalFrc module 62
Figure 4.7. a3DTrussStrs module 63
Figure 4.8. Result object for 3D Truss structures 64
Figure 4.9. Flowchart of the 3D Truss program. 65
Figure 4.10. Plane truss problem 66
Figure 4.11. 3D Truss problem 69
Figure 5.1. Plane frame examples 72
Figure 5.2. Internal forces of a beam 72
Figure 5.3. Beam element 73
Figure 5.4. Beam3Stiff module. 75
Figure 5.5. PBGlobStrStiff module 76
Figure 5.6. PBReducedNodeForces module 78
Figure 5.7. PBSolution module. 79
Figure 5.8. PBIntForces module 81
Figure 5.9. Result2 object for beam-column frames 82
Figure 5.10. Flowchart of the beam-column frame program 83

Figure 5.11. Beam-column problem 84
Figure 6.1. Types of quadrilateral element 88
Figure 6.2. Local coordinate system for quadrilateral 89
Figure 6.3. Gauss1D module 96
Figure 6.4. Schematic representation of 1D Gauss rule 97
Figure 6.5. Gauss2D module 98
Figure 6.6. Schematic representation of 2D Gauss rule 99
Figure 6.7. The 4 point quadrilateral element 101
Figure 6.8. Stiffness matrix of 4 point quadrilateral element 101
Figure 6.9. 4 point quadrilateral element example 103
Figure 6.10. The 8 point quadrilateral element 105
Figure 6.11. Stiffness matrix of 8 point quadrilateral element 106
Figure 6.12. The 9 point quadrilateral element 108
Figure 6.13. Stiffness matrix of 9 point quadrilateral element 109
Figure 7.1. Area coordinates of a triangle 112
Figure 7.2. Pnt3TrgMembStiff module 115
Figure 7.3. Triangular Gauss points 116
Figure 7.4. Triangular Gauss module 117
Figure 7.5. The 6 point isoparametric triangle element 120
Figure 7.6. Pnt6TrgMembStiff module 121
Figure 7.7. Pnt6TrgJacobi module 122
Figure 8.1. Examples of plane stress problems 123
Figure 8.2. PSGlobStrStiff module 125
Figure 8.3. PSSolution module 127
Figure 8.4. Pnt4QdrMembStresses module 129

Figure 8.5. Pnt8QdrMembStresses module 130

Figure 8.6. Pnt9QdrMembStresses module 131

Figure 8.7. Pnt3TrgMembStresses module 132

Figure 8.8. Pnt6TrgMembStresses module 133

Figure 8.9. StressAveraging module 135

Figure 8.10. Results object for plane stress problems 137

Figure 8.11. Flowchart of the plane stress program 138

Figure 9.1. Cantilever beam plane stress problem 139

Figure 9.2. Local coordinates for exact solution formulas 140

Figure 9.3. Triangular mesh node points of cantilever beam case 142

Figure 9.4. Quadrilateral mesh node points of cantilever beam case 143

Figure 9.5. δ_y Accuracy of triangular elements at point A. 147

Figure 9.6. δ_y Accuracy of quadrilateral elements at point A 148

Figure 9.7. δ_x Accuracy of triangular elements at point A. 150

Figure 9.8. δ_x Accuracy of quadrilateral elements at point A 151

Figure 9.9. δ_y Accuracy of triangular elements at point D. 153

Figure 9.10. δ_y Accuracy of quadrilateral elements at point D. 154

Figure 9.11. δ_x Accuracy of triangular elements at point D. 156

Figure 9.12. δ_x Accuracy of quadrilateral elements at point D 157

Figure 9.13. τ_{xy} Accuracy of triangular elements at point C 159

Figure 9.14. τ_{xy} Accuracy of quadrilateral elements at point C 160

Figure 9.15. σ_x Accuracy of triangular elements at point D 162

Figure 9.16. σ_x Accuracy of quadrilateral elements at point D. 163

Figure 9.17. σ_x Accuracy of triangular elements at point E. 165

Figure 9.18. σ_x Accuracy of quadrilateral elements at point E 166

Figure 10.1. Simply supported beam plane stress problem 167

Figure 10.2. Local coordinates for exact solution formulas 168

Figure 10.3. Quadrilateral mesh of simply supported beam case 170

Figure 10.4. Contours of Ansys δ_y solution. 172

Figure 10.5. Contours of Java δ_y solution. 172

Figure 10.6. Contours of Ansys δ_x solution. 173

Figure 10.7. Contours of Java δ_x solution. 173

Figure 10.8. Contours of Ansys σ_x solution 174

Figure 10.9. Contours of Java σ_x solution. 174

Figure 10.10. Contours of Ansys σ_y solution 175

Figure 10.11. Contours of Java σ_y solution 175

Figure 10.12. Contours of Ansys τ_{xy} solution 176

Figure 10.13. Contours of Java τ_{xy} solution 176

LIST OF TABLES

Table 4.1. Properties of the plane truss problem 66
Table 4.2. Input data of the plane truss problem 67
Table 4.3. Comparison of element stresses 67
Table 4.4. Comparison of displacements 68
Table 4.5. Properties of the 3D Truss problem 68
Table 4.6. Input data of the 3D Truss problem 69
Table 4.7. Comparison of element stresses 70
Table 4.8. Comparison of displacements 71
Table 5.1. Properties of the beam frame problem 85
Table 5.2. Comparison of internal forces of beam elements 86
Table 5.3. Comparison of node deflections 87
Table 6.1. Abscissa and weights for Gaussian quadrature rules 95
Table 8.1. Element classification. 124
Table 9.1. Properties of the plane truss problem 139
Table 9.2. The exact values of stresses and deflections 141
Table 9.3. Input file names given in Appendix A for cantilever beam case 144
Table 9.4. Comparison of vertical deflection at point A 146
Table 9.5. Comparison of accuracies of vertical deflection at point A 146
Table 9.6. Comparison of horizontal deflection at point A 149
Table 9.7. Comparison of accuracies of horizontal deflection at point A 149
Table 9.8. Comparison of vertical deflection at point D. 152
Table 9.9. Comparison of accuracies of vertical deflection at point D 152
Table 9.10. Comparison of horizontal deflection at point D 155

Table 9.11. Comparison of accuracies of horizontal deflection at point D 155

Table 9.12. Comparison of shear stress at point C 158

Table 9.13. Comparison of accuracies of shear stress at point C 158

Table 9.14. Comparison of normal stress at point D 161

Table 9.15. Comparison of accuracies of normal stress at point D 161

Table 9.16. Comparison of normal stress at point E 164

Table 9.17. Comparison of accuracies of normal stress at point E 164

Table 10.1. Properties of the plane truss problem 167

Table 10.2. The exact values of stresses 169

Table 10.3. Comparison tables of the stresses and deflections 171

Table A.1. Number of Gauss points for rank-sufficient plane stress elements 181

LIST OF SYMBOLS/ABBREVIATIONS

{A}	Gradient operator
{e}	Strain matrix
{u}	Displacement matrix
{w}	Displacement matrix of beam element
A	Cross-section area
a_o,.. a_i	Constants of the polynomials
B	Strain-displacement matrix
D	Stress-strain matrix
E	Modulus of elasticity
f’	Node forces of local system
f_1,.. f_i	End forces
f^e	Element forcing vector
f_e	Node forces of global system
fj (x)	Continuous functions
f_{x1},.. f_{xi}	End node forces
$f_{x1},\ldots,f_{xi}$	Node horizontal forces
$f_{y1},\ldots,f_{yi}$	Node vertical forces
G	Shear modulus
h	Height
I	Moment of inertia
J	Jacobian matrix
k’	Local stiffness matrix
k^e	Global element stiffness matrix
K^G	Global structural stiffness matrix
k_m	Element stiffness matrix
L	Length
M	Bending moment
N	Shape function matrix
N_i (x)	Shape functions
P	Force
P_n(x)	Legendre polynomials

q	Uniform transverse load
R(x)	Residual
T	Displacement transformation matrix
t	Thickness
u(x)	Exact solution of differential equation
u’	Displacements of local system
u† (x)	Approximated solution of differential equation
u_1,.. u_i	End displacements
u^e	Displacements of global system
$u_{x1},\ldots,u_{xi}$	Node horizontal displacements
$u_{y1},\ldots,u_{yi}$	Node vertical displacements
w_i	Gaussian quadrature rule weights
Π	Stored potential energy
ξ_1, η_2, ζ_3	Area coordinates
ξ, η	Local natural coordinates
α	Angle between local and global coordinates
ν	Poisson ratio
σ	Element stresses
θ	Rotation angle
Δd	Relative displacement
δU	Strain energy
δW	Work done by external loads
JRE	Java Runtime Environment
CPU	Central Processing Unit

1. INTRODUCTION TO THE FINITE ELEMENT METHOD AND THE JAVA LANGUAGE

1.1. Historical Background of the Finite Element Method

Many physical problems in engineering and science can be described in terms of partial differential equations. Generally, solving these equations by classical analytical methods for arbitrary shapes is almost impossible. These partial differential equations can be approximately solved by numerical algorithms [1]. Recently, numerical methods have been used for various technical disciplines in order to predict behavior of structural, mechanical, thermal, fluid flow and electrical systems.

Numerical methods used today are at least as old as the Egyptian Rhind papyrus which describes a root-finding method for solving a simple equation. Ancient Greek mathematician Archimedes made much further advancement in numerical methods between 285–212 BC by calculating lengths, areas, and volumes of geometric figures. The pioneers of the modern numerical integration are Isaac Newton and Gottfried Leibniz. Calculus of Isaac Newton led to accurate mathematical models for physical reality, first in the physical sciences and eventually in the engineering, medicine, and business. Although these mathematical models are usually too complicated, the numerical methods created by Newton are used for solving variety of problems, such as finding solutions for general functions or finding a polynomial equation that best fits a set of data. Following Newton, well known mathematicians of the 18^{th} and 19^{th} centuries, the Swiss Leonhard Euler, the French Joseph-Louis Lagrange and the German Carl Friedrich Gauss made major contributions to numerical analysis. Applying these developments to the basic laws of physics described by Newton, many mathematical models for solid and fluid mechanics are obtained. Civil engineers and mechanical engineers still base their models on these obtained results. Subsequently, in the 19th century heat, electricity, and magnetism were successfully modeled and finally, in the 20th century, relativistic mechanics, quantum mechanics, and other theoretical constructs were created by further improvements [2].

One of the most widespread numerical analysis techniques for solving partial differential equations related with the engineering problems is the finite element method. Recently, finite element method is widely used for solving engineering problems such as stress analysis, heat transfer, fluid flow and electromagnetic coupling by computer simulation. Finite element method was outlined by the mathematician Richard Courant in 1943 [3] and developed by the American engineer Harold Martin and others to analyze stress forces on new jet wing design of the Boeing Company in the 1950s [4].

The basic idea in the finite element method is to find the solution of a complicated problem by dividing it into simple elements. To explain the basic approach of the finite element method, consider a steel plate in plane stress under uniform uniaxial loading in x direction with a central circular hole as shown in Figure 1.1. Here, in order to obtain an approximate solution, the domain to be analyzed is divided into cells, or elements connected by nodes. This is called the finite element mesh and the process is called mesh generation. Since displacement field is continuous across element boundaries, the strain at every point can be expressed in terms of nodal displacements. It is required that the stresses associated with these strains, through the stress-strain relations of the material, satisfy the principle of virtual work for arbitrary variation of the nodal displacements. Therefore, compatibility and force equilibrium equations can be written for each point in the plate. For linear problems, the solution is determined by solving these linear equations in which the number of unknowns is equal to the number of nodes. To obtain a reasonably accurate solution, thousands of nodes are usually needed, so computers are essential for solving these equations [5].

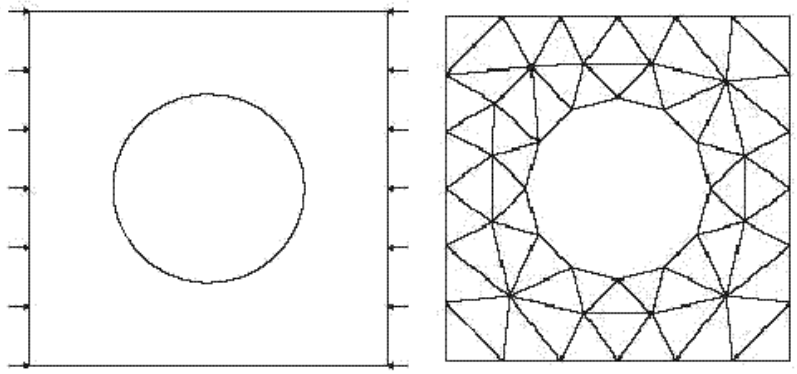


Figure 1.1. Finite element meshing

1.2. Computer Science Revolution

Finite element method is one of the most widely adopted computational techniques used in computer science. Although most prestigious journals were refused to publish papers on finite element method in the beginning, in the late 1960s it has been shown that as the number of finite elements increases, exact solution accuracy of the partial differential equations is improved. Therefore, finite element is become popular for solving structural and mechanical systems. Subsequently, several computer implementations are issued.

In 1965, NASA funded a project to develop a general-purpose finite element program known as NASTRAN which includes a large array of capabilities, such as two- and three-dimensional stress analyses, beam and shell elements, for analyzing complex structures. Just after NASA, in 1969, a private cooperation Westinghouse Electric developed a program called ANSYS which had linear and nonlinear capabilities. Finally, ABAQUS was introduced by a company called HKS, which was founded in 1978. With the rapid growth in capabilities of computer hardware, importance of the software design and implementation process became even more significant. Finite element codes were generally developed in Fortran but during last decade developers started to seek better programming paradigms in order to handle complexity in finite element software [1].

Programming languages vary from statically typed languages like C to dynamically typed languages like python and different programming paradigms are present such as object-oriented like Java and procedural like Fortran. Software quality is assessed based on many different aspects; for instance code readability, software design complexity, robustness, run time performance and portability. Design phase of the software affects many of the aspects of the final product. With the introduction of the object oriented programming paradigm, the design process of software is simplified in that the real world problem can be mapped onto its abstraction in a more natural way. Using the object-oriented approach with data hiding, encapsulation and inheritance, allows creating reliable and extensible finite element codes.

1.3. The Birth of the Java Language

Java is an object-oriented programming language released by Sun Microsystems in 1995. It is a static-typed language so that Java compiler at compile time checks code to impose the static rules. One main advantage of Java over its counterpart C++ is the lack of operator overloading and multiple inheritance in Java. This, seeming as a shortcoming, is actually an advantage since it greatly simplifies the language and prevents ambiguity thereby making the final source code more generic. Generic and simple code means that further developments can be easily made by different people on the software since ambiguity is not allowed by the Java language.

Java is a C based language and it is possible to integrate libraries written in C with Java code since there are ways to interface C object codes within Java. This ability makes programming with Java much easier, given the vast array of libraries coded in C which one can make use of. Java also supports the open source community and source code of Java's core is open source. Furthermore, Java is a simple language with rich collection of libraries implementing various application programming interfaces. While object-oriented programming can be done with Java, it also has built-in garbage collector that prevents memory leaks. Other useful features such as actual portability, flexibility and presentation are unique characteristics of Java language [6].

1.4. The Advantages of the Java Language

1.4.1. Platform Independency

The one of the most important aspect of Java is that it is a platform-independent programming language. A Java code is compiled into what is called a bytecode which is an intermediate form of the executable. Then bytecodes are supplied to a Java Runtime Environment and JRE runs the program.

JRE is actually a piece of software tailored for the specific hardware it will run on, which serves as a platform for the Java bytecode to run on. Since software industry is going around porting applications to different platforms, platform independency is an

invaluable asset for programmers. Porting is the process in which source code of an executable is modified so that the resultant executable can run on a target platform that is different from the one for which the original application was written. Using Java makes the porting process very easy, most of the time not requiring any changes to the original source code. Java is designed to be able to run the same code on different operating systems and hardware and also supply built-in functionality to accomplish tasks involving networking.

1.4.2. Object Oriented Programming Paradigm

From the standing point of software architecture, object-oriented programming paradigm when applied to certain programming tasks makes life much easier than with a procedural approach. The object-oriented approach maps onto the definition of the programming task in a smooth way. In classic procedural programming, the units of modularity can be identified as a set of functions. However in object oriented programming, the modules can be viewed as objects that are interrelated. Furthermore, an object has its own data and its own methods which both can have access specifiers restricting their accessibility from outside the object thus defining a clear interface of what an object means to other objects. Functions of an object can act on data members of that object, the data members and the functionality defined on top of the data reside together in one unit called the object.

Objects are entities that live throughout the run-time of a software. The class is the definition of an object, like a composite data type for which certain functions can be implemented which will operate on the data. To put it in another way, an object is an instantiation of a class; in a way, an object is related to its class like the way a variable is related to its type. Classes reduce the redundancy stemming from the code by providing modularity in such a way that the real world problem is mapped onto the object oriented paradigm. The testing and finding bugs in an object oriented program is much simpler because there are well defined interfaces between classes and there are structural properties imposed by classes such as data encapsulation and information hiding. The data members of classes are called fields and the functionality that is implemented on top of those fields are called methods. Both fields and methods have access specifiers that allow only a subset of fields/methods of a class to be accessible from outside the class. This ability to hide

certain information when necessary greatly reduces bugs and simplifies testing since some constraints are imposed by the compiler, for instance one can not change the value of a certain field from within another class scope if that field was declared as private [7].

1.4.3. Simplicity of Programming

Another advantage of Java over procedural languages like Fortran and C lies in the ease of programming. Java does not have the concept of pointers as in C which can be a real challenge, in that most of the programming effort is spent on preventing memory leaks manually and taking care of address manipulation.

In Java, one does not need to take care of memory maintenance since there is a garbage collector imposed by the runtime environment automatically and a powerful exception handling mechanism to handle undefined behavior. The garbage collector will deallocate any memory space preserved for objects which can no longer be referred that is, for instance the object's scope has finished. Normally, in C, when the programmer has used an array for representing a matrix, he/she should free the memory allocated to that matrix; on the other hand, a Java programmer need not worry about deallocating the memory since it is handles automatically by the system, in other words Java has the automated memory management sub-system so that programmers do not have to take care of the memory related issues manually. Besides, Java has the strict exception handling mechanism which aids the programmer in reducing bugs by not allowing any undefined behavior to exist in the program flow. Garbage collection, automatic memory management and powerful exception handling properties make Java programs less prone to bugs and saves the programmer time.

1.4.4. Built-in Multithreading

Java has another useful ability which is the support for multithreading at the language level. Other languages such as the famous C and Fortran do not have native multithreading support, 3rd party libraries need to be used to achieve multithreading. Threads are execution paths inside the program, on multi-processor architectures using threads greatly improve performance, and if the task is inherently parallel like in matrix

multiplication then the performance gain is huge. Java provides certain thread classes to utilize multi-threaded programs in a unified and structured way.

1.4.5. Flexibility and Presentation

As a scientist, by programming in Java one need not worry about operating system or platform specific issues. For instance, a Fortran program is to be written for a Linux machine greatly differs from the same program written for a windows machine. In order to address many different platforms the programmer needs to have enough experience in each platform and also needs to watch and use the different platforms to be able to update necessary knowledge of platform specific issues. However, by using Java the scientist has to learn the Java platform and that is it, Java platform can run on other platforms and there is no need to be expert on each targeted specific platform.

Most of the time, a scientific program which performs miracles under its interface is undervalued due to the lack of charm in its presentation. And the scientist is not the person to spend most of his/her time on making the application look good, rather a scientist is to concentrate on the functionality of the application, whether it is performing its task correctly or not and within a limited time budget. Java handles that part in a standard way so that the programmer does not waste much time dealing with the presentation of the final product [8].

1.5. Performance of the Java Language

In an ideal world, programs assumed to run directly on top of the hardware without generating any overhead but it is not the case in a real world. Since hardware specifications change rapidly and program codes have to be rewritten every time a part of hardware is changed, directly interfacing with the hardware is not possible. However, through the operating system, JRE provides the direct interface to the hardware so that it takes care of hardware changes.

Finite element methods mostly depend on computing power utilized in numerical algorithms such as matrix operations. JRE is designed such that the CPU time for making

pure mathematical operations is greatly optimized, so in fact there is no CPU power lost when it comes to numerical methods. Besides, with the growing popularity of Java, the JRE also evolved and there are many optimizations involved which improves the performance of Java codes especially with the programs which numerical methods are implemented.

Finally, considering its performance, portability and ease of programming, it is possible to conclude that the Java language is suitable for development of finite element software with the use of proper coding. Therefore, in this study Java language is chosen for the implementation of one dimensional and two dimensional engineering problems.

2. FUNDAMENTALS OF THE FINITE ELEMENT METHOD

2.1. Mathematical Theory

2.1.1. Galerkin Method of Weighted Residuals

In mathematics, in the area of numerical analysis, Galerkin methods are a class of methods for converting an operator problem such as a differential equation to a discrete problem. In principle, it is the equivalent of applying the method of variation to a function space, by converting the equation to a weak formulation.

The following differential equation to be solved over a one dimensional domain *D*, subject to some boundary condition

$$p\frac{d^2u}{dx^2}+q=0 \tag{2.1}$$

where *p* and *q* are constants, and $a \leq x \leq b$. By using the method of weighted residuals the exact solution u(x) can be approximated by $u^{\dagger}(x)$, where

$$u(x) \approx u^{\dagger}(x)=\sum_{j=1}^{m}\phi_j(x)c_j \tag{2.2}$$

the trial functions," ϕ_j (x)", are continues functions which are satisfying boundary conditions of equation (2.1) and c_j are unknown parameters. When equation (2.2) is substituted into equation (2.1), the equation becomes

$$p\frac{d^2u^{\dagger}}{dx^2}+q=R(x) \tag{2.3}$$

where *R(x)* is a measure of the error in the approximation and is called residual. The residual will vanish for the exact solution. In the Galerkin weighted residual method, a

weighted average of this residual is required to vanish over the domain of the equation also the weighting functions are chosen to be the trial functions.

$$\int \phi_j(x) R(x) dx = \int \left[p \frac{d^2 u^\dagger}{dx^2} + q \right] \phi_j(x) dx = 0 \tag{2.4}$$

In order to process Galerkin method, domain is separated into seven discrete elements as shown in figure 2.1.

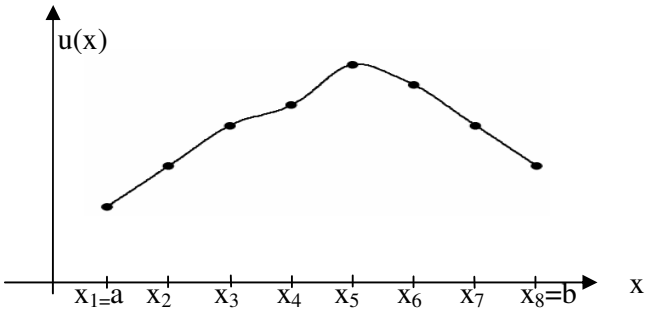


Figure 2.1. Finite element representation of u(x)

By applying equation (2.4) to each discrete element the new equation becomes;

$$\int_{x_i}^{x_{i+1}} \phi_j dx = \int_{x_i}^{x_{i+1}} \left[p \frac{d^2 u^\dagger}{dx^2} + q \right] \phi_j(x) dx = 0 \tag{2.5}$$

Since the only conditions restricting trial functions are continuity and boundary satisfaction, ϕ_j can be chosen to be polynomials of low degree. If polynomials are chosen than $u^\dagger(x)$ may be written as

$$u^\dagger(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots \tag{2.6}$$

where the a_i are unknown constants. For a linear variation $u^\dagger(x)$ becomes;

$$u^\dagger(x) = a_0 + a_1 x \tag{2.7}$$

Part of the finite element model of u (x) is the set of its nodal values. These nodal values can clearly be used to determine the unknown coefficients a_i in equation (2.6). Considering second element over u(x), the coefficients a_i *are* solved by the following system of equations.

$$\begin{aligned} u_2 &= a_0 + a_1 x_2 \\ u_3 &= a_0 + a_1 x_3 \end{aligned} \tag{2.8}$$

Than $u^{\dagger}(x)$ can be written as

$$u^{\dagger}(x) = \frac{x - x_2}{x_2 - x_3} u_2 + \frac{x - x_2}{x_3 - x_2} u_3 \tag{2.9}$$

Eventually by matching equations (2.9) and (2.2); m, c_j and ϕ_j (x) are found as

$$\begin{aligned} m &= 2 \\ c_1 = u_2 \quad &\text{and} \quad c_2 = u_3 \\ \phi_1 = \frac{x - x_3}{x_2 - x_3} \quad &\text{and} \quad \phi_2 = \frac{x - x_2}{x_3 - x_2} \end{aligned} \tag{2.10}$$

In the finite element literature, these ϕ_j (x) functions are denoted by N_i (x) and generally known as shape functions. Since the final product of the finite element method is the production of a system of algebraic equations, at the outset matrix notation will be used. In matrix form equation (2.9) becomes

$$u^{\dagger} = \begin{bmatrix} N_1, N_2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \tag{2.11}$$

Substituting equation (2.11) into equation (2.5) produces the following function

$$\int_{x_2}^{x_3} \begin{bmatrix} N_1 \\ N_2 \end{bmatrix} p \frac{d^2}{dx^2} \begin{bmatrix} N_1, N_2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} dx + \int_{x_2}^{x_3} \begin{bmatrix} N_1 \\ N_2 \end{bmatrix} q dx = 0 \tag{2.12}$$

Since double differentiation of the shape functions (2.10) would cause them to vanish, Green's integration by parts theorem will be applied to the shape functions. Thus integration can be written as

$$\int_{x_2}^{x_3} N_i \frac{d^2 N_j}{dx^2} = -\int_{x_2}^{x_3} \frac{dN_i}{dx}\frac{dN_j}{dx}dx + \left[N_i \frac{dN_j}{dx} \right]_{x_2}^{x_3} \tag{2.13}$$

The boundary terms that occur on inter-element boundaries will cancel each other since the combination of the element solutions is essentially additive. For example on the boundary between second and third elements, equation (2.13) for each will produce the following terms:

$$N_i \frac{dN_j}{dx}\bigg|_{x_4} - N_i \frac{dN_j}{dx}\bigg|_{x_3} + N_i \frac{dN_j}{dx}\bigg|_{x_3} - N_i \frac{dN_j}{dx}\bigg|_{x_2} \tag{2.14}$$

The only parts of these boundary terms that will be left are those actually on the boundary of the domain [a, b]. Hence for second element, equation (2.12) becomes

$$p\int_{x_2}^{x_3} \begin{bmatrix} \frac{dN_1}{dx}\frac{dN_1}{dx} & \frac{dN_1}{dx}\frac{dN_2}{dx} \\ \frac{dN_2}{dx}\frac{dN_1}{dx} & \frac{dN_2}{dx}\frac{dN_2}{dx} \end{bmatrix} dx \begin{bmatrix} u_2 \\ u_3 \end{bmatrix} - q\int_{x_2}^{x_3} \begin{bmatrix} N_1 \\ N_2 \end{bmatrix} dx = 0 \tag{2.15}$$

Employing the integrations, the equations can be expressed as

$$p\begin{bmatrix} \frac{1}{L} & -\frac{1}{L} \\ -\frac{1}{L} & \frac{1}{L} \end{bmatrix} \begin{bmatrix} u_2 \\ u_3 \end{bmatrix} - q\begin{bmatrix} \frac{L}{2} \\ \frac{L}{2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{2.16}$$

where $L= x_3 - x_2$. This becomes in matrix notation

$$\mathrm{k}_m \mathrm{u} - \mathrm{F}^e = 0 \tag{2.17}$$

where k_m is the element stiffness matrix and F^e is the element forcing vector [9].

2.2. Derivation of One Dimensional Element Stiffness

The simplest structural member in finite element method is the "2 node rod element". Rod elements constitute plane trusses such as the one depicted in Figure 2.2. Plane truss structures are formed by structural components called rod and beam members which are connected at joints. Since longitudinal dimension of these members is significantly larger than the other two dimensions, they are classified as one dimensional members.

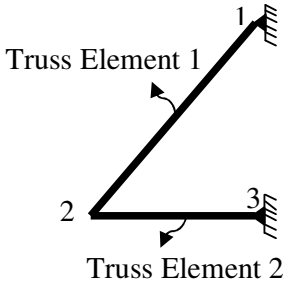


Figure 2.2. Sample plane truss

2.2.1. The Rod Element Stiffness

The simplest solid element in structural mechanics is a rod element which is subjected to axial loadings along its longitudinal axis.

Figure 2.3 shows an elastic rod with end nodes 1 and 2. The element which has a length of L is subjected to an axial load of P. F_1 and F_2 are the end forces while u(x) is the longitudinal displacements of points on the rod.

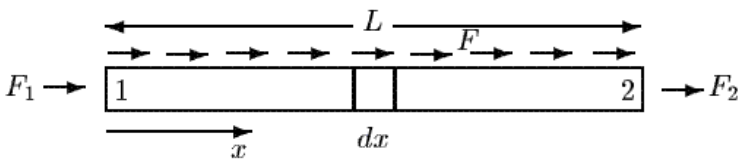


Figure 2.3. Rod element

Figure 2.4 shows the equilibrium of the rod partition subjected to a uniform longitudinal body force F.

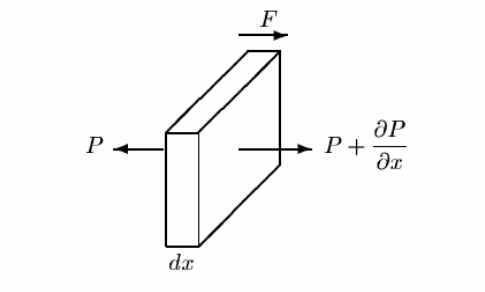


Figure 2.4. Rod element equilibrium

Since small longitudinal strain is just the rate of change of longitudinal displacement, axial stress σ is linearly related to strain by Young's modulus E.

$$P = \sigma A = EA\varepsilon = EA\frac{\partial u}{\partial x} \tag{2.18}$$

Assuming small strain, the equilibrium equation can be written as

$$\frac{\partial P}{\partial x} + F = 0 \tag{2.19}$$

hence the differential equation to be solved is:

$$EA\frac{\partial^2 u}{\partial x^2} + F = 0 \tag{2.20}$$

The solution of this differential equation is given in equation (2.16). By replacing p by EA and q by F in the equation (2.16), the following equation is derived.

$$EA\begin{bmatrix} \frac{1}{L} & -\frac{1}{L} \\ -\frac{1}{L} & \frac{1}{L} \end{bmatrix}\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} - F\begin{bmatrix} \frac{L}{2} \\ \frac{L}{2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{2.21}$$

The above case is for a uniformly distributed force F acting along the element, and it should be noted that the Galerkin procedure has resulted in the total force FL being shared

equally between the two nodes. If the load is applied only at the nodes, the equation becomes

$$\frac{EA}{L}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}\begin{bmatrix} u_1 \\ u_2 \end{bmatrix}=\begin{bmatrix} f_{x1} \\ f_{x2} \end{bmatrix} \tag{2.22}$$

where f_{x1} is the force in the x-direction at node 1 and f_{x2} is the force in the x-direction at node 2.

Equation (2.22) represents the rod element stiffness relationship. It is indicated in matrix notation as [9]

$$[k_m]\{u\} = \{f\} \tag{2.23}$$

2.2.2. The Beam Element Stiffness

Structural elements which resist transverse loads mainly through bending action along its longitudinal dimension are called beam elements.

The beam element shown in Figure 2.5 with end nodes 1 and 2 has length L, flexural rigidity EI, and carries a uniform transverse load of q. The end nodes are subjected to shear forces and moments which cause translations and rotations. Therefore, each node has two degrees of freedom.

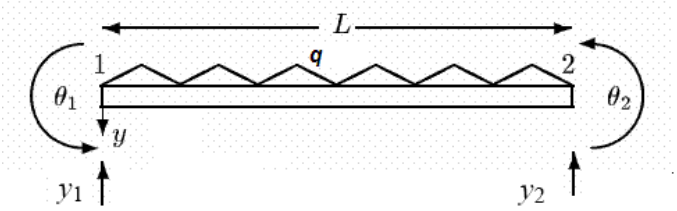


Figure 2.5. Beam element

Equilibrium equation for this system is given by

$$EI\frac{\partial^4 y}{\partial x^4} = q \tag{2.24}$$

Again the continuous variable, y in this case, is approximated in terms of discrete nodal values; however here not only y itself but also its derivatives θ must be used in the approximation. In this case the continuous variable y is approximated by $y^{\dagger}$ in terms of nodal values as follows:

$$y^{\dagger} = \begin{bmatrix} N_1 & N_2 & N_3 & N_4 \end{bmatrix} \begin{bmatrix} y_1 \\ \theta_1 \\ y_2 \\ \theta_2 \end{bmatrix} = [N]\{w\} \tag{2.25}$$

Furthermore, the cubic shape functions are chosen to satisfy boundary conditions and their derivatives in this case, equal one at a specific node and zero at all others.

$$\begin{aligned} N_1 &= (L^3 - 3Lx^2 + 2x^3)/L^3 \\ N_2 &= (L^2x - 2Lx^2 + x^3)/L^2 \\ N_3 &= (3Lx^2 - 2x^3)/L^3 \\ N_4 &= (x^3 - Lx^2)/L^2 \end{aligned} \tag{2.26}$$

Substitution of equations (2.25) in equation (2.24) and than applying Galerkin method leads to the four basic equations:

$$\int_0^L \begin{bmatrix} N_1 \\ N_2 \\ N_3 \\ N_4 \end{bmatrix} EI \frac{\partial^4}{\partial x^4} \begin{bmatrix} N_1 & N_2 & N_3 & N_4 \end{bmatrix} \begin{bmatrix} y_1 \\ \theta_1 \\ y_2 \\ \theta_2 \end{bmatrix} dx = \int_0^L \begin{bmatrix} N_1 \\ N_2 \\ N_3 \\ N_4 \end{bmatrix} q \, dx \tag{2.27}$$

In order to avoid differentiating four times, Green Theorem is used :

$$\int N_i \frac{\partial^4 N_j}{\partial x^4} dx = \int \frac{\partial N_i}{\partial x} \frac{\partial^3 N_j}{\partial x^3} dx = -\int \frac{\partial^2 N_i}{\partial x^2} \frac{\partial^2 N_j}{\partial x^2} dx \tag{2.28}$$

Since EI and q are not function of x, left hand side of equation (2.27) becomes

$$EI\int_0^L \left[\frac{\partial^2 N_i}{\partial x^2}\frac{\partial^2 N_j}{\partial x^2}\right] dx \begin{bmatrix} y_1 \\ \theta_1 \\ y_2 \\ \theta_2 \end{bmatrix} \tag{2.29}$$

Evaluation of the integral gives:

$$\frac{EI}{L^3}\begin{bmatrix} 12 & 6L & -12 & 6L \\ 6L & 4L^2 & -6L & 2L^2 \\ -12 & -6L & 12 & -6L \\ 6L & 2L^2 & -6L & 4L^2 \end{bmatrix} \begin{bmatrix} y_1 \\ \theta_1 \\ y_2 \\ \theta_2 \end{bmatrix} = \frac{q}{12}\begin{bmatrix} 6L \\ L \\ 6L \\ -L \end{bmatrix} \tag{2.30}$$

which recovers the standard "slope-deflection" equations for beam elements. Equation (2.30) is evaluated for a uniformly distributed load applied to the beam. For the case in which loading is applied only at nodes, the following equation is obtained;

$$\frac{EI}{L^3}\begin{bmatrix} 12 & 6L & -12 & 6L \\ 6L & 4L^2 & -6L & 2L^2 \\ -12 & -6L & 12 & -6L \\ 6L & 2L^2 & -6L & 4L^2 \end{bmatrix} \begin{bmatrix} y_1 \\ \theta_1 \\ y_2 \\ \theta_2 \end{bmatrix} = \begin{bmatrix} f_{z1} \\ m_1 \\ f_{z2} \\ m_2 \end{bmatrix} \tag{2.31}$$

which represents the beam element stiffness relationship.

Hence, in matrix notation it becomes

$$[k_m]\{\mathrm{w}\} = \{\mathrm{f}\} \tag{2.32}$$

which has the same form as equation (2.23) and is the standard form of all equilibrium problems formulated in terms of finite elements [9].

2.2.3. Derivation of One Dimensional Element Stiffness by Energy Approach

Instead of working from governing differential equation, elements stiffness properties can be expressed by using the virtual work principle. Virtual work principle may be described as when a relatively small set of arbitrary displacements, compatible with the kinematical and geometrical boundary condition, are given to a system in equilibrium under external loads, the increase in work done by the external loads over these small displacements, equals the increase in the strain energy of the system [10].

The principle can be expressed as

$$\delta U = \delta W \tag{2.33}$$

The strain energy stored due to bending of a very small length δx of the elastic beam element in Figure 2.5 is,

$$\delta U = \frac{1}{2}\frac{M^2}{EI}\delta x \tag{2.34}$$

where M is the "bending moment" and EI is flexural rigidity. The work done by the external load q is equals to

$$\delta W = \frac{1}{2} qy\delta x \tag{2.35}$$

The bending moment M is related to y through the moment-curvature expression

$$M = -EI\frac{d^2y}{dx^2} \tag{2.36}$$

or in matrix form;

$$M = [\mathrm{D}]\{\mathrm{A}\}\, y \tag{2.37}$$

Where $[D]$ is the material property EI and {A} is the operator $-d^2/dx^2$. By rearranging equation (2.34) in the form,

$$\delta U = \frac{1}{2}\left(-\frac{d^2 y}{dx^2}\right) M \delta x \tag{2.38}$$

than the equation becomes

$$\delta U = \frac{1}{2}\left(\{\mathrm{A}\}\, y\right)^{\mathrm{T}} M \delta x \tag{2.39}$$

Substituting equation (2.25) into equation (2.39) by applying shape function relationships, following equation is obtained.

$$\begin{aligned} \delta U &= \frac{1}{2}\left(\{\mathrm{A}\}[\mathrm{N}]\{\mathrm{w}\}\right)^{\mathrm{T}} [\mathrm{D}]\{\mathrm{A}\}[\mathrm{N}]\{\mathrm{w}\}\delta x \\ &= \frac{1}{2}\{\mathrm{w}\}^{\mathrm{T}} \left(\{\mathrm{A}\}[\mathrm{N}]\right)^{\mathrm{T}} [\mathrm{D}]\{\mathrm{A}\}[\mathrm{N}]\{\mathrm{w}\}\delta x \end{aligned} \tag{2.40}$$

The total strain energy of the element is thus,

$$U = \frac{1}{2}\int_0^L \{\mathrm{w}\}^{\mathrm{T}} \left(\{\mathrm{A}\}[\mathrm{N}]\right)^{\mathrm{T}} [\mathrm{D}]\{\mathrm{A}\}[\mathrm{N}]\{\mathrm{w}\}\mathrm{d}x \tag{2.41}$$

Shortly the product $[A][N]$ can be written as $[B]$. Since $\{\mathrm{w}\}$ are nodal values and they are constants, the final equation is derived as follows

$$U = \frac{1}{2}\{\mathrm{w}\}^{\mathrm{T}} \int [\mathrm{B}]^{\mathrm{T}} [\mathrm{D}][\mathrm{B}]\mathrm{d}x\{\mathrm{w}\} \tag{2.42}$$

Similar operations on equation (2.34) lead to the external work done equals to

$$W = \frac{1}{2}\{\mathrm{w}\}^{\mathrm{T}} q \int_0^L [\mathrm{N}]^{\mathrm{T}} \mathrm{d}x \tag{2.43}$$

Finally, stored potential energy of the beam is given by

$$\begin{aligned}\prod &= U - W \\ &= \frac{1}{2}\{\mathrm{w}\}^{\mathrm{T}} \int_0^L [\mathrm{B}]^{\mathrm{T}} [\mathrm{D}][\mathrm{B}]\mathrm{d}x\{\mathrm{w}\} - \frac{1}{2}\{\mathrm{w}\}^{\mathrm{T}} q\int_0^L [\mathrm{N}]^{\mathrm{T}}\mathrm{d}x\end{aligned} \tag{2.44}$$

A state of stable equilibrium is achieved when Π is a minimum with respect to all {w},

$$\frac{\partial \prod}{\partial \{\mathrm{w}\}^{\mathrm{T}}} = \int_0^L [\mathrm{B}]^{\mathrm{T}} [\mathrm{D}][\mathrm{B}]\mathrm{d}x\{\mathrm{w}\} - q\int_0^L [\mathrm{N}]^{\mathrm{T}}\mathrm{d}x = 0 \tag{2.45}$$

or

$$\int_0^L [\mathrm{B}]^{\mathrm{T}} [\mathrm{D}][\mathrm{B}]\mathrm{d}x\{\mathrm{w}\} = q\int_0^L [\mathrm{N}]^{\mathrm{T}}\mathrm{d}x \tag{2.46}$$

which is the same expression of equation (2.27).

Consequently, the element stiffness matrix $[k_m]$ can be written in the form [11].

$$[\mathrm{k}_m] = \int_0^L [\mathrm{B}]^{\mathrm{T}} [\mathrm{D}][\mathrm{B}]\mathrm{d}x \tag{2.47}$$

2.3. Master Stiffness Method

Matrix Stiffness Method is the most common structural analysis method which is particularly suited for computer-automated linear static analysis of complex structures. Matrix stiffness method, utilizes matrices and matrix algebra to organize the structural system as a set of simpler elements interconnected at the nodes. Matrices which are subjected to specific rules can be used to achieve the solution of unknown displacements and forces of the system.

2.3.1. Introduction

Application of the master stiffness method of structural analysis requires subdividing the structure into a set of finite elements, where the endpoints are called nodes. For the case of trusses, each truss member is considered as a finite element, and each joint

becomes a node. Once the elements are identified, each element is then analyzed individually to develop member stiffness equations in their local coordinates. After localization is performed, globalization of stiffness matrix is required to institute the assembly process which is technically a process of converting the stiffness relations for the individual elements into a global system for the entire structure.

Global Structural Stiffness Matrix $[K]$ which contains force-displacement relationship for entire structure is used to solve all unknown displacements for any given loading scenario. Subsequently, after displacements are known, all unknown reaction forces can be determined.

2.3.2. Element Stiffness Matrix in Local Coordinates

In order to define element stiffness matrix of a single truss member in its own local coordinate system, an inclined rod element is sketched in Figure 2.6.

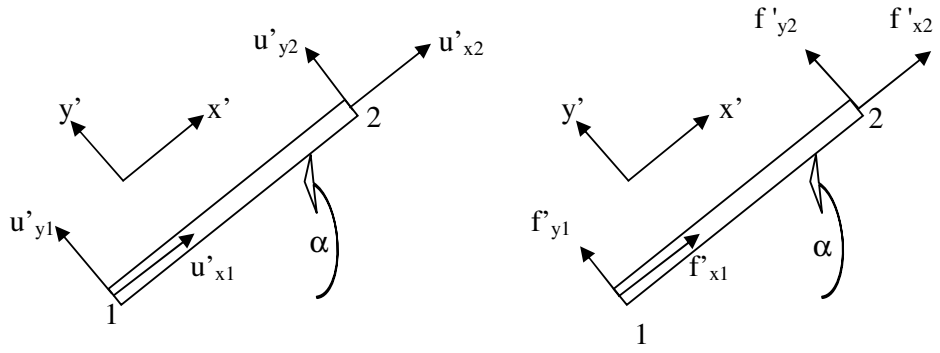


Figure 2.6. End displacements and end forces

The coordinates of sketched local system are {x',y'} and the two end joints of the chosen member are 1 and 2. The sketched truss member has four joint displacement (u'_{x1}, u'_{y1}, u'_{x2}, u'_{y2}) and four force components (f'_{x1}, f'_{y1}, f'_{x2}, f'_{y2}). These four components are also known as degrees of freedom. The member has a length of L, elastic modulus E and cross-section area A.

The force and displacement relationship of a rod element under axial loadings is evaluated in equation (2.22), which is repeated here for convenience:

$$\frac{EA}{L}\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}\begin{bmatrix} u_1 \\ u_2 \end{bmatrix}=\begin{bmatrix} f_{x1} \\ f_{x2} \end{bmatrix} \tag{2.48}$$

Since sketched rod element has four degrees of freedom, force-displacement relations are expanded as follows

$$[K']\{u'\}=\frac{EA}{L}\begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} u'_{x1} \\ u'_{y1} \\ u'_{x2} \\ u'_{y2} \end{bmatrix}=\begin{bmatrix} f'_{x1} \\ f'_{y1} \\ f'_{x2} \\ f'_{y2} \end{bmatrix}=\{f'\} \tag{2.49}$$

2.3.3. Element Stiffness Matrix in Global Coordinates

Since a truss structure consists of many members of possibly many orientations, all local stiffness equations have to be transformed into a single uniform global coordinate system. This transformation process is evaluated by connecting joint displacements and forces in the global and local coordinate systems.

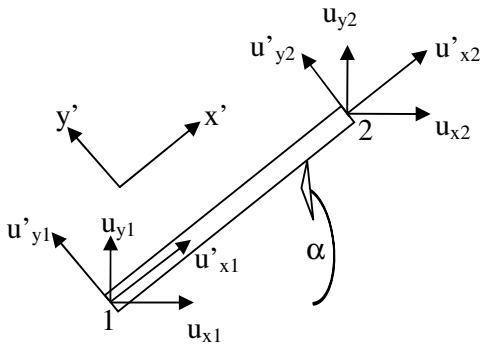


Figure 2.7. Node displacement transformations

Figure 2.7 shows an individual element with the local displacements (u'_{x1}, u'_{y1}, u'_{x2}, u'_{y2}), and the global displacements (u_{x1}, u_{y1}, u_{x2}, u_{y2}). The transformation of node displacement components from local system (x',y') to global system (x,y) is obtained by following angular correlations.

$$u'_{x1} = u_{x1}\cos\alpha + u_{y1}\sin\alpha \;,\; u'_{y1} = -u_{x1}\sin\alpha + u_{y1}\cos\alpha$$
$$u'_{x2} = u_{x2}\cos\alpha + u_{y2}\sin\alpha \;,\; u'_{y2} = -u_{x2}\sin\alpha + u_{y2}\cos\alpha \qquad (2.50)$$

where α is the angle between x an x' measured positive counterclockwise from x. These correlations can be written as;

$$\begin{bmatrix} u'_{x1} \\ u'_{y1} \\ u'_{x2} \\ u'_{y2} \end{bmatrix} = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & \cos\alpha & \sin\alpha \\ 0 & 0 & -\sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} u_{x1} \\ u_{y1} \\ u_{x2} \\ u_{y2} \end{bmatrix} \qquad (2.51)$$

The matrix which interconnects local displacements to global displacements is called displacement transformation matrix and denoted by T. Since displacements $(u_{x1}, u_{x2}, u_{y1}, u_{y2})$ are in global coordinate system matrix notation becomes

$$u' = Tu^e \qquad (2.52)$$

Here u^e indicates that displacements are in terms of global system. The transformation of node force components which are depicted in Figure 2.8 can be obtained by the similar angular correlations.

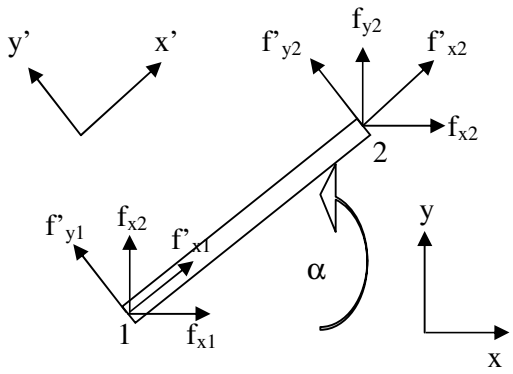


Figure 2.8. Node force transformations

$$f_{x1} = f'_{x1}\cos\alpha - f'_{y1}\sin\alpha \ , \ f_{y1} = f'_{x1}\sin\alpha + f'_{y1}\cos\alpha \tag{2.53}$$

$$f_{x2} = f'_{x2}\cos\alpha - f'_{y2}\sin\alpha \ , \ f_{y2} = f'_{x2}\sin\alpha + f'_{y2}\cos\alpha \tag{2.54}$$

where α is the same angle between x and x' coordinates, measured positive counterclockwise from x. Node force transformation between local and global coordinates can be written in matrix form in equation (2.55).

$$\begin{bmatrix} f_{x1} \\ f_{y1} \\ f_{x2} \\ f_{y2} \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & \cos\alpha & -\sin\alpha \\ 0 & 0 & \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} f'_{x1} \\ f'_{y1} \\ f'_{x2} \\ f'_{y2} \end{bmatrix} \tag{2.55}$$

Since the transformation matrix in Equation (2.55) interconnects local node forces to global node forces, it is called force transformation matrix. By visual inspection, it can be showed that force transformation matrix is the transpose of displacement transformation matrix and it is denoted by T^T [12]. Hence, in matrix notation it becomes

$$f^e = T^T f' \tag{2.56}$$

Here f^e indicates that node forces are in global system where f' indicates local node forces. Subsequently, the transformation matrices derived from angular correlations can be used to define the stiffness matrix in terms of global coordinates. Displacement force relation in local coordinates is given in equation (2.57).

$$f' = k'u' \tag{2.57}$$

Substituting equation (2.52) into (2.57) gives

$$f' = k'T\ u^e \tag{2.58}$$

When both side of the equation (2.58) is multiplied by T^T, it becomes

$$T^T f' = T^T k' T \, u^e \tag{2.59}$$

Finally, equation (2.56) is substituted into equation (2.59) and the equation becomes

$$f^e = T^T k' T \, u^e \tag{2.60}$$

Since f^e and u^e are the displacement and force matrices of the global system, remaining terms represent a member stiffness matrix in global coordinates which is expressed in equation (2.61).

$$k^e = T^T k' T \tag{2.61}$$

Carrying out the matrix multiplications in equation (2.61) the following equation is obtained.

$$k^e = \frac{E^e A^e}{L^e} \begin{bmatrix} \cos^2\alpha & \sin\alpha\cos\alpha & -\cos^2\alpha & -\sin\alpha\cos\alpha \\ \sin\alpha\cos\alpha & \sin^2\alpha & -\sin\alpha\cos\alpha & -\sin^2\alpha \\ -\cos^2\alpha & -\sin\alpha\cos\alpha & \cos^2\alpha & \sin\alpha\cos\alpha \\ -\sin\alpha\cos\alpha & -\sin^2\alpha & \sin\alpha\cos\alpha & \sin^2\alpha \end{bmatrix} \tag{2.62}$$

2.3.4. Assembly of Global Structural Stiffness Matrix

Equation derived in equation (2.62) relates the displacements to forces, all in global coordinates, for a single element of arbitrary orientations. The contribution of a single element is accounted for the element global stiffness matrix [k^e]. For a structure with multiple members, it is required to assemble the global element stiffness matrix of each member. It is simply a process of adding each into a global structural stiffness matrix that covers every degree of freedom in the entire structure. Global structural stiffness matrix [K^G] is formed as a square matrix with as many rows and columns as there are total degrees of freedom. In order to explain the assembly process briefly a sample structure is used which is sketched in Figure 2.9.

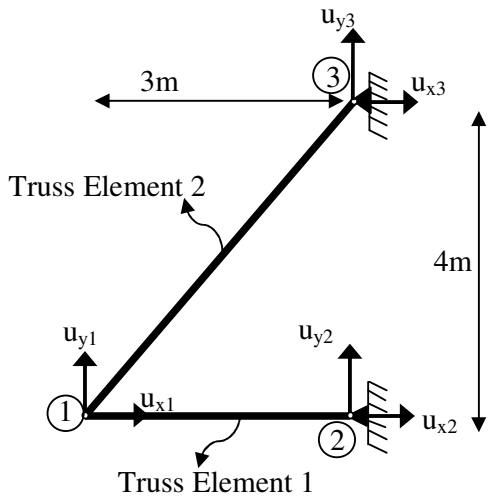


Figure 2.9. Sample structure

In the sketched figure two interconnected members have 6 degrees of freedom. For each degree of freedom there is an associated u_x and u_y displacements. Before the assembly process, the global member stiffness of each element is needed to be calculated by using equation (2.62).

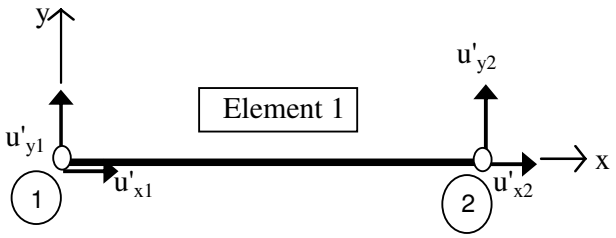


Figure 2.10. Sample element 1

The element labeled 1, connects nodes 1 and 2 as shown in Figure 2.10. The local displacements in node 1 are u_{x1} and u_{y1}, and the local displacements in node 2 are u_{x2} and u_{y2}. The length of the element is equal to 3 meters and $\sin\alpha = 0, \cos\alpha = 1$. By using equation (2.62) k_1^e can be evaluated as;

$$k_1^e = AE \begin{array}{c} \begin{matrix} u_{x1} & u_{y1} & u_{x2} & u_{y2} \end{matrix} \\ \begin{bmatrix} 0.333 & 0 & -0.333 & 0 \\ 0 & 0 & 0 & 0 \\ -0.333 & 0 & 0.333 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{array} \begin{matrix} u_{x1} \\ u_{y1} \\ u_{x2} \\ u_{y2} \end{matrix} \tag{2.63}$$

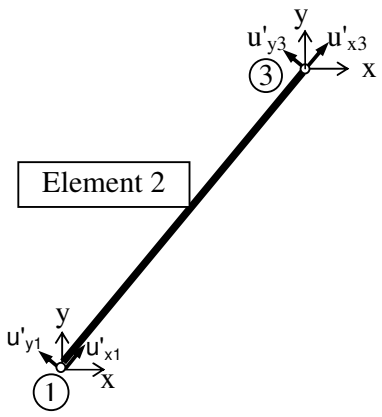


Figure 2.11. Sample element 2

The element labeled 2 connects nodes 1 and 3 as shown in Figure 2.11. The local displacements in node 1 are u_{x1} and u_{y1}, and the local displacements in node 3 are u_{x3} and u_{y3}. The length of the element is equal to five meters and $\sin\alpha = 0.8, \cos\alpha = 0.6$. By using equation (2.62) k_2^e can be evaluated as;

$$k_2^e = AE \begin{matrix} \\ u_{x1} \\ u_{y1} \\ u_{x3} \\ u_{y3} \end{matrix} \begin{matrix} \begin{matrix} u_{x1} & u_{y1} & u_{x3} & u_{y3} \end{matrix} \\ \begin{bmatrix} 0.072 & 0.096 & -0.072 & -0.096 \\ 0.096 & 0.128 & -0.096 & -0.128 \\ -0.072 & -0.096 & 0.072 & 0.096 \\ -0.096 & -0.128 & 0.096 & 0.128 \end{bmatrix} \end{matrix} \tag{2.64}$$

Now all element stiffness matrices are evaluated. In order to construct global structural stiffness matrix, it is required to add each related row and column into the zero matrix which has 6x6 dimensions.

$$K^G = AE \begin{matrix} \\ u_{x1} \\ u_{y1} \\ u_{x2} \\ u_{y2} \end{matrix} \begin{matrix} \begin{matrix} u_{x1} & u_{y1} & u_{x2} & u_{y2} \end{matrix} \\ \begin{bmatrix} 0.333 & 0 & -0.333 & 0 \\ 0 & 0 & 0 & 0 \\ -0.333 & 0 & 0.333 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{matrix} + AE \begin{matrix} \\ u_{x1} \\ u_{y1} \\ u_{x3} \\ u_{y3} \end{matrix} \begin{matrix} \begin{matrix} u_{x1} & u_{y1} & u_{x3} & u_{y3} \end{matrix} \\ \begin{bmatrix} 0.072 & 0.096 & -0.072 & -0.096 \\ 0.096 & 0.128 & -0.096 & -0.128 \\ -0.072 & -0.096 & 0.072 & 0.096 \\ -0.096 & -0.128 & 0.096 & 0.128 \end{bmatrix} \end{matrix} \tag{2.65}$$

By summing k_1^e and k_2^e, the global structural stiffness matrix becomes

$$K^G = AE \begin{array}{c} \begin{matrix} u_{x1} & u_{y1} & u_{x2} & u_{y2} & u_{x3} & u_{y3} \end{matrix} \\ \begin{bmatrix} 0.405 & 0.096 & -0.333 & 0 & -0.072 & -0.096 \\ 0.096 & 0.128 & 0 & 0 & -0.096 & -0.128 \\ -0.333 & 0 & 0.333 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -0.072 & -0.096 & 0 & 0 & 0.072 & 0.096 \\ -0,096 & -0.128 & 0 & 0 & 0.096 & 0.128 \end{bmatrix} \end{array} \begin{matrix} u_{x1} \\ u_{y1} \\ u_{x2} \\ u_{y2} \\ u_{x3} \\ u_{y3} \end{matrix} \qquad (2.66)$$

2.3.5. Reduced Global Structural Stiffness System Calculations

Having formed the global structural stiffness matrix also known as master stiffness matrix, full system displacement-force relations become;

$$\begin{bmatrix} f_{x1} \\ f_{y1} \\ f_{x2} \\ f_{y2} \\ f_{x3} \\ f_{y3} \end{bmatrix} = AE \begin{bmatrix} 0.405 & 0.096 & -0.333 & 0 & -0.072 & -0.096 \\ 0.096 & 0.128 & 0 & 0 & -0.096 & -0.128 \\ -0.333 & 0 & 0.333 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -0.072 & -0.096 & 0 & 0 & 0.072 & 0.096 \\ -0,096 & -0.128 & 0 & 0 & 0.096 & 0.128 \end{bmatrix} \begin{bmatrix} u_{x1} \\ u_{y1} \\ u_{x2} \\ u_{y2} \\ u_{x3} \\ u_{y3} \end{bmatrix} \qquad (2.67)$$

In equation (2.67), the displacements (u_{x1}, u_{y1}, u_{x2}, u_{y2}, u_{x3}, $u_{y3)}$ are the unknowns in global coordinates system, and (f_{x1}, f_{y1}, f_{x2}, f_{y2}, f_{x3}, f_{y3}) are the external loads.

Since the rows and columns of K^G are linear combinations of each other, K^G is a singular matrix. Therefore, equation (2.67) could not be solved. The physical meaning of this singularity is that the truss system is unsuppressed at its supports which make it move along (x,y) plane.

In order to avoid this movement, support conditions have to be applied to the system. These boundary conditions can be applied by removing associated equations from the global structural stiffness matrix. This can be accomplished by deleting rows and columns of u_{x2}, u_{y2}, u_{x3}, u_{y3} and corresponding components of force matrix.

$$\left[\begin{array}{c} f_{x1} \\ f_{y1} \\ \hdashline f_{x2} \\ f_{y2} \\ f_{x3} \\ f_{y3} \end{array}\right] = AE \left[\begin{array}{cc:cccc} 0.405 & 0.096 & -0.333 & 0 & -0.072 & -0.096 \\ 0.096 & 0.128 & 0 & 0 & -0.096 & -0.128 \\ \hdashline -0.333 & 0 & 0.333 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -0.072 & -0.096 & 0 & 0 & 0.072 & 0.096 \\ -0,096 & -0.128 & 0 & 0 & 0.096 & 0.128 \end{array}\right] \left[\begin{array}{c} u_{x1} \\ u_{y1} \\ \hdashline u_{x2} \\ u_{y2} \\ u_{x3} \\ u_{y3} \end{array}\right] \tag{2.68}$$

The reduced two equation system is called reduced master stiffness system which is stated in equation (2.69).

$$\left[\begin{array}{c} f_{x1} \\ f_{y1} \end{array}\right] = AE \left[\begin{array}{cc} 0.405 & 0.096 \\ 0.096 & 0.128 \end{array}\right] \left[\begin{array}{c} u_{x1} \\ u_{y1} \end{array}\right] \tag{2.69}$$

2.3.6. Solution for the Unknown Displacements

Since singularity of the master stiffness matrix is prevented by applying displacement boundary conditions, unknown displacements can be solved by using equation (2.70). The solution process of the sample structure can be generalized for all structures by using the following matrix operations.

$$[f_{reduced}] = [K_{reduced}][u_{unknown}] \tag{2.70}$$

Assuming, ten unit force in the negative y direction at node 1.

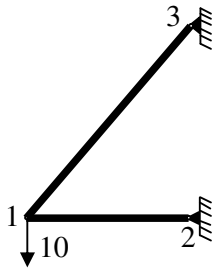


Figure 2.12. Sample external load

Reduced global structural system of the sample structure becomes;

$$\begin{bmatrix} 0 \\ -10 \end{bmatrix} = AE \begin{bmatrix} 0.405 & 0.096 \\ 0.096 & 0.128 \end{bmatrix} \begin{bmatrix} u_{x1} \\ u_{y1} \end{bmatrix} \tag{2.71}$$

Solving equation (2.71) for u_{x1}, u_{y1} gives the displacement solution.

$$\begin{aligned} u_{x1} &= +22.52 / EA \\ u_{y1} &= -95.02 / EA \end{aligned} \tag{2.72}$$

2.3.7. Calculations of Reaction Forces

All information required for the calculation of mechanical quantities is evaluated in the Master Stiffness Method procedure.

Since global structural stiffness matrix accommodates the global forces and global displacement relations directly, multiplying K^G with complete displacement matrix gives global forces acting on the structure. For the sample structure, multiplication of K^G and displacement solution matrix is given in equation (2.73).

$$\begin{bmatrix} f_{x1} \\ f_{y1} \\ f_{x2} \\ f_{y2} \\ f_{x3} \\ f_{y3} \end{bmatrix} = AE \begin{bmatrix} 0.405 & 0.096 & -0.333 & 0 & -0.072 & -0.096 \\ 0.096 & 0.128 & 0 & 0 & -0.096 & -0.128 \\ -0.333 & 0 & 0.333 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -0.072 & -0.096 & 0 & 0 & 0.072 & 0.096 \\ -0,096 & -0.128 & 0 & 0 & 0.096 & 0.128 \end{bmatrix} \begin{bmatrix} +22.5 / EA \\ -95 / EA \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ -10 \\ -7.5 \\ 0 \\ 7.5 \\ 10 \end{bmatrix} \tag{2.73}$$

From equation (2.73), the reaction forces of f_{x2}, f_{y2}, f_{x3}, f_{y3}, can be evaluated. Thus, reaction forces of sample structure are sketched in Figure 2.13.

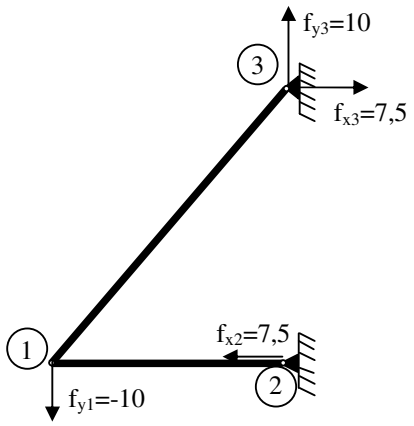


Figure 2.13. Reaction forces of the sample structure

Note that, while displacements u_{x1} and u_{x2} are calculated in terms of the material properties AE, the reactions are found as numerical values. Thus, force equilibrium is not dependent on material properties, while displacements are directly dependent on material properties.

2.3.8. Calculations of Internal Forces and Stresses

The aim of structural design is to determine the dimensions and the internal quantities of the system. Since the most essential quantities for the structural design are the internal forces, they have to be calculated in the various members of the framework. In the case of a rod element, the only internal forces are the axial forces and stresses which are sketched in the Figure 2.14.

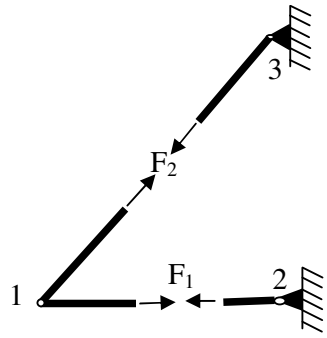


Figure 2.14. Internal forces of the sample structure

In physics, Hooke's law of elasticity refers that the amount by which a material body is deformed is linearly proportional to the force causing the deformation. Thus, internal force of an element can be evaluated by;

$$F = \frac{EA}{L}\Delta d \tag{2.74}$$

Where E is elastic modules, A is cross-section area and Δd is the relative displacement of the member edge nodes. Relative displacement of a member in its local coordinates is defined by the equation (2.75) as follows

$$\Delta d = u'_{x2} - u'_{x1} \tag{2.75}$$

In order to calculate relative displacement of the member edge nodes, it is required to transform global displacements into local displacements. The transformation matrix derived in equation (2.52) is repeated here for convenience:

$$\mathrm{u}' = \mathrm{T}\mathrm{u}^{\mathrm{e}} \tag{2.76}$$

By substituting the associated displacement solutions into equation (2.76), the local displacements of element 2 are evaluated.

$$\begin{bmatrix} u'_{x1} \\ u'_{y1} \\ u'_{x3} \\ u'_{y3} \end{bmatrix} = \begin{bmatrix} 0.6 & 0.8 & 0 & 0 \\ -0.8 & 0.6 & 0 & 0 \\ 0 & 0 & 0.6 & 0.8 \\ 0 & 0 & -0.8 & 0.6 \end{bmatrix} \begin{bmatrix} +22.5/EA \\ -95/EA \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -62.5/EA \\ -75/EA \\ 0 \\ 0 \end{bmatrix} \tag{2.77}$$

The local displacements derived in equation (2.77) are substituted into the equation (2.75) and relative displacements of the element 2 nodes are calculated.

$$\begin{aligned} u'_{x1} &= -62.5/EA \\ u'_{x2} &= 0 \\ \Delta d &= 62.5/EA \end{aligned} \tag{2.78}$$

Since the relative displacement is calculated, the numerical value of the axial force of the element 2 can be derived from equation (2.74) as follows.

$$F_2 = \frac{EA}{L}\Delta d = \frac{EA}{5}.\frac{62.5}{EA} = 12.5 \tag{2.79}$$

The evaluation of the internal force for the sample rod element can be generalized by using the following matrix multiplication [12].

$$[f'] = [k'][T]\left[u^e\right] \tag{2.80}$$

Applying the general matrix multiplication equation, axial force of element 1 in sample structure can be calculated as follows:

$$F_1 = -7.5 \tag{2.81}$$

Negative sign in equation (2.81), indicates that element 1 is under compression. Finally, average axial stress σ is obtained by dividing axial forces by the cross-sectional area of the member. Axial stresses of element 1 and element 2 are equal to;

$$\begin{aligned} \sigma_1 &= -\frac{7.5}{A} \\ \sigma_2 &= \frac{12.5}{A} \end{aligned} \tag{2.82}$$

2.4. Derivation of Two Dimensional Element Stiffness

Many problems in engineering can be treated satisfactorily by plane theory of elasticity which uses two dimensional continuum finite elements. Two major types of plane analysis problems are plane stress and plane strain. Plane stress is defined to be a state of stress in which the normal stress, σ_z, and the shear stresses, σ_{xz} and σ_{yz}, directed to the x-y plane are assumed to be zero. On the other hand, plane strain is defined to be a state of strain in which the strain normal to the x-y plane, ε_z, and the shear strains γ_{xz} and

γ_{yz}, are assumed to be zero. By taking two-dimensional state of stress into account, in a plane stress problem, three independent stresses, $\sigma_x, \sigma_y, \tau_{xy}$ and three independent strains $\varepsilon_{xx}, \varepsilon_{yy}, \gamma_{xy}$ forms two vectors denoted as $\{\sigma\}$ and $\{\varepsilon\}$, respectively [13].

Displacement matrix {u}, defined by u_x and u_y which are the components of displacement in the x and y directions. Subsequently, f_x and f_y are the force components that constitute {f} matrix.

If thickness of the given domain is assumed to be constant than the equilibrium, constitutive and strain-displacement equations to be solved are the following:

$$\begin{aligned} \frac{\partial \sigma_x}{\partial x} + \frac{\partial \tau_{xy}}{\partial y} + F_x = 0 \\ \frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \sigma_y}{\partial y} + F_y = 0 \end{aligned} \tag{2.83}$$

$$\begin{Bmatrix} \sigma_x \\ \sigma_y \\ \tau_{xy} \end{Bmatrix} = \frac{E}{1-\upsilon^2} \begin{bmatrix} 1 & \upsilon & 0 \\ \upsilon & 1 & 0 \\ 0 & 0 & \dfrac{1-\upsilon}{2} \end{bmatrix} \begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \tau_{xy} \end{Bmatrix} \tag{2.84}$$

$$\begin{Bmatrix} \varepsilon_x \\ \varepsilon_y \\ \tau_{xy} \end{Bmatrix} = \begin{bmatrix} \dfrac{\partial}{\partial x} & 0 \\ 0 & \dfrac{\partial}{\partial y} \\ \dfrac{\partial}{\partial y} & \dfrac{\partial}{\partial x} \end{bmatrix} \begin{Bmatrix} u_x \\ u_y \end{Bmatrix} \tag{2.85}$$

Equations (2.83), (2.84), (2.85) can be written in the form [11];

$$\begin{aligned} [A]^T \{\sigma\} &= -\{f\} \\ \{\sigma\} &= [D]\{\varepsilon\} \\ \{\varepsilon\} &= [A]\{u\} \end{aligned} \tag{2.86}$$

where

$$[A]=\begin{bmatrix} \frac{\partial}{\partial x} & 0 \\ 0 & \frac{\partial}{\partial y} \\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x} \end{bmatrix}, [D]=\frac{E}{1-\upsilon^2}\begin{bmatrix} 1 & \upsilon & 0 \\ \upsilon & 1 & 0 \\ 0 & 0 & \frac{1-\upsilon}{2} \end{bmatrix} \tag{2.87}$$

By arranging equations (2.86), the following equation is derived;

$$[A]^T[D][A]\{e\}=-\{f\} \tag{2.88}$$

Hence the partial differential equation to be solved is;

$$\frac{E}{1-v^2}\begin{bmatrix} \frac{\partial^2 u}{\partial x^2}+\frac{(1-v)}{2}\frac{\partial^2 u}{\partial y^2}+\frac{(1+v)}{2}\frac{\partial^2 v}{\partial x\partial y} \\ \frac{(1+v)}{2}\frac{\partial^2 u}{\partial x\partial y}+\frac{(1-v)}{2}\frac{\partial^2 v}{\partial x^2}+\frac{\partial^2 v}{\partial y^2} \end{bmatrix}=\begin{bmatrix} -F_x \\ -F_y \end{bmatrix} \tag{2.89}$$

Discretisation over element domains using shape functions and application of Galerkin's method leads to the stiffness equations for a typical element. The stiffness relationship can be written in the standard form of equation (2.23) [11].

$$[k_m]\{u\}=\{f\} \tag{2.90}$$

2.4.1. Derivation of Two Dimensional Element Stiffness by Energy Approach

The principle of minimum potential energy which provides an alternative derivation of one dimensional element stiffness matrix derived in equation (2.47) can be generalized for two dimensional case.

The element strain energy per unit thickness is

$$\begin{aligned} U &= \iint \frac{1}{2}\{\sigma\}^T \{\varepsilon\}\, dxdy \\ U &= \frac{1}{2}\{u\}^T \iint ([A][N])^T [D]([A][N])dxdy\{u\} \\ U &= \frac{1}{2}\{u\}^T \iint [B]^T [D][B]dxdy\{u\} \end{aligned} \tag{2.91}$$

where $[A]$ and $[D]$ are defined in equation (2.87). By taking;

$$[N] = \begin{bmatrix} N_1 & 0 & N_2 & 0 & N_3 & 0 & N_4 & 0 \\ 0 & N_1 & 0 & N_2 & 0 & N_3 & 0 & N_4 \end{bmatrix} \tag{2.92}$$

and applying $[B] = [A][N]$, $[B]$ matrix leads to;

$$[B] = \begin{bmatrix} \frac{\delta N_1}{\delta x} & 0 & \frac{\delta N_2}{\delta x} & 0 & \frac{\delta N_3}{\delta x} & 0 & \frac{\delta N_4}{\delta x} & 0 \\ 0 & \frac{\delta N_1}{\delta y} & 0 & \frac{\delta N_2}{\delta y} & 0 & \frac{\delta N_3}{\delta y} & 0 & \frac{\delta N_4}{\delta y} \\ \frac{\delta N_1}{\delta y} & \frac{\delta N_1}{\delta x} & \frac{\delta N_2}{\delta y} & \frac{\delta N_2}{\delta x} & \frac{\delta N_3}{\delta y} & \frac{\delta N_3}{\delta x} & \frac{\delta N_4}{\delta y} & \frac{\delta N_4}{\delta x} \end{bmatrix} \tag{2.93}$$

Here, $[B]$ matrix is called the stain displacement matrix.

Therefore, element stiffness matrix in equation (2.46) is expended for two dimensional elements as follows [11].

$$[k_m] = \iint [B]^T [D][B]dxdy \tag{2.94}$$

3. JAVA IMPLEMENTATION OF MASTER STIFFNESS METHOD

A Plane truss is a structural system composed of members designed to sustain only axial loads. In reality, the members in a truss plane system encounter with a small amount of bending and twisting, however the members are assumed to have no bending and torsion resistance.

All members of a theoretical truss are supposed to be connected at their ends by frictionless connections. The center axis of each member should intersect exactly at a common point so members do not permit bending moment to be transferred through the joint. Several examples of simple 2-dimensional plane trusses are depicted in Figure 3.1.

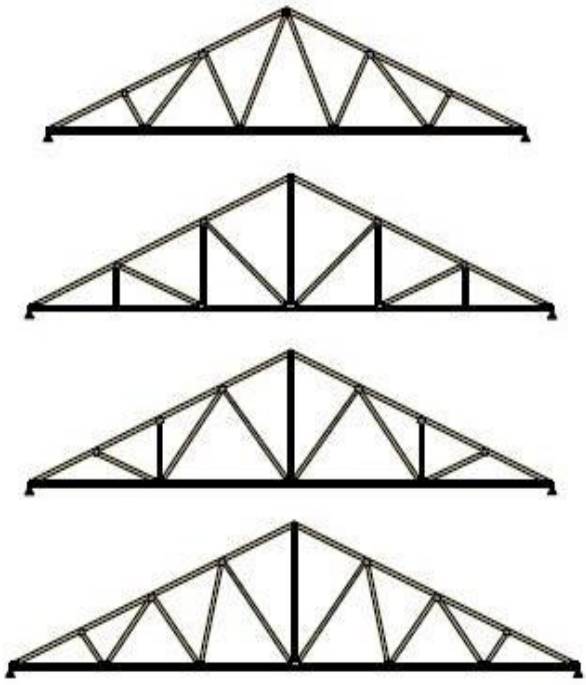


Figure 3.1. Plane truss examples

3.1. Implementation of Global Element Stiffness Matrix

The Java code shown in Figure 3.3 called Rod2Stiff returns global stiffness matrix of a rod element which has two degrees of freedom at each nodes.

3.1.1. Formal Parameters of Rod2Stiff:

Coordinates: Global coordinates of the nodes of the rod element {(x1,y1),(x2,y2)}

ElasticM: Elastic modulus of the rod element (E)

area: Cross-section area of the rod element (A)

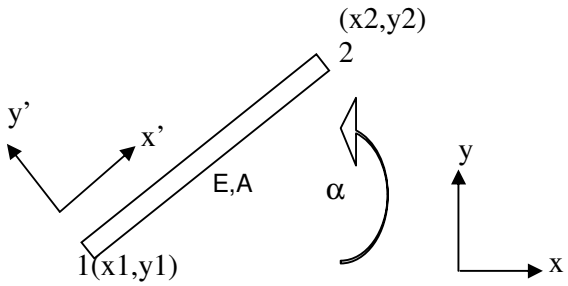


Figure 3.2. Rod element in local coordinates

In order to implement global stiffness matrix of the rod element which is evaluated in equation (2.62), length of the rod, sinus and cosines of "α" are required.

In analytic geometry, the distance between two points (x1, y1) and (x2, y2) of the xy-plane is given by

$$L = \sqrt{(\Delta x)^2 + (\Delta y)^2} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{3.1}$$

Local variable x21 is assigned for Δx which is the difference between x_2 and x_1. Local variable y21 is assigned for Δy which is the difference between y_2 and y_1. When analytic relations are substituted into equation (2.62), the global stiffness matrix equation becomes

$$k^e = \frac{E^e A^e}{(L^e)^3} \begin{bmatrix} \Delta x^2 & \Delta x \Delta y & -\Delta x^2 & -\Delta x \Delta y \\ \Delta x \Delta y & \Delta y^2 & -\Delta x \Delta y & -\Delta y^2 \\ -\Delta x^2 & -\Delta x \Delta y & \Delta x^2 & \Delta x \Delta y \\ -\Delta x \Delta y & -\Delta y^2 & \Delta x \Delta y & \Delta y^2 \end{bmatrix} \tag{3.2}$$

Equation (3.2) is directly implemented in Java module called Rod2Stiff which is shown in Figure 3.3.

```
public static double[][] Rod2Stiff(double [][]Coordinates, double
ElasticM, double area)
   {
      double x21=Coordinates[1][0]-Coordinates[0][0];
      double y21=Coordinates[1][1]-Coordinates[0][1];
      double factor=
ElasticM*area/(Math.pow(Math.sqrt(x21*x21+y21*y21),3));
      double [][] ke=
         {{ x21*x21, x21*y21,-x21*x21,-x21*y21},
         { y21*x21, y21*y21,-y21*x21,-y21*y21},
         {-x21*x21,-x21*y21, x21*x21, x21*y21},
         {-y21*x21,-y21*y21, y21*x21, y21*y21}};
      for(int i=0;i<ke.length;i++)
      {
         for(int j=0;j<ke[i].length;j++)
            ke[i][j]=ke[i][j]*factor;
      }
      return ke;
   }
```

Figure 3.3. Rod2Stiff module

3.1.2. Example Commands for Rod2Stiff:

The global stiffness matrix of the rod members of the simple structure illustrated in Figure 3.4 can be evaluated by the following commands.

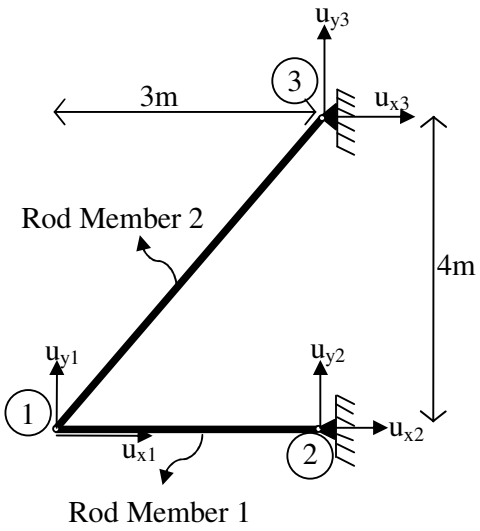


Figure 3.4. Rod2Stiff example

"Rod2Stiffness [{(0,0),(3,0)},{E},{A}] " command calls the first rod member with global end node coordinates of [(0,0),(3,0)], Elastic modulus of E and cross-section area of A. These numerical values are assigned to the formal parameters of the module and it returns the global stiffness matrix of the rod element 1. (k_1^e) given in equation (3.3).

$$k_1^e = AE\begin{bmatrix} 0.333 & 0 & -0.333 & 0 \\ 0 & 0 & 0 & 0 \\ -0.333 & 0 & 0.333 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \tag{3.3}$$

On the other hand, "Rod2Stiffness [{(0,0),(3,4)},E,A]" command calls the second rod member with global end node coordinates of [(0,0),(3,4)], Elastic modulus of E and cross-section area of A. These numerical values are assigned to the formal parameters of the module and it returns the global stiffness matrix of the rod element 2. (k_2^e) given in equation (3.4).

$$k_2^e = AE\begin{bmatrix} 0.072 & 0.096 & -0.072 & -0.096 \\ 0.096 & 0.128 & -0.096 & -0.128 \\ -0.072 & -0.096 & 0.072 & 0.096 \\ -0.096 & -0.128 & 0.096 & 0.128 \end{bmatrix} \tag{3.4}$$

3.2. Implementation of Global Structural Stiffness Matrix

The Java code shown in Figure 3.5 called PTGlobStrStiff returns global structural matrix of any defined plane truss system.

3.2.1. Formal Parameters of PTGlobStrStiff

entirecoord: All global coordinates of the system joints
{(x1,y1),(x2,y2)…….(x_{last}, y_{last})}

elemposition: Element end points that are assigned for the system joints which are required to define an element and its local axis.
{(#1, #2)$_1$, (#1, #2)$_2$….. (#1, #2)$_{last}$}

ElasticMs: Elastic Modulus of each element in the same order of "elemposition" parameter (E_1,E_2….E_{last})

areas: Cross-sections of each element in the same order of "elemposition" parameter A_1,A_2….A_{last})

Global structural stiffness matrix, "K^G" is a square matrix which accommodates all displacement force relations of every degree of freedoms. Since plane truss elements have two freedoms in every node, the total number of the freedom is equal to;

$$\Sigma n_{freedom} = \text{total number of freedom } = 2 * \text{ number of nodes} \tag{3.5}$$

Thus, global structural stiffness matrix has ($\Sigma n_{freedom} * \Sigma n_{freedom}$) dimensions. In the first step of implementation, total number of freedom is computed by reading the length of the formal parameter entirecoord and multiplying it with two. Subsequently, module constructs ($\Sigma n_{freedom} * \Sigma n_{freedom}$) dimensional zero matrix.

$$\overbrace{\begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix}}^{\Sigma n_{freedom}} \Bigg\{ \Sigma n_{freedom} \tag{3.6}$$

In the second step, global end node coordinates of all members are assigned by using the formal parameter elemposition. Local variable "xycoord" contains global end node coordinates of each member. Since all required formal parameters are assigned for each member, Rod2Stiff module is called. Rod2Stiff module returns, global stiffness matrix of each member. Local variable "littlek" contains each k_n^e of the system.

In the final step, each k_n^e of the system is merged into the global structural matrix. Each related row and column of the k_n^e are defined by the local variable "ongfi". Local variable "ongfi" contains, order number of the global freedom indices in related member.

```
public static double[][] PTGlobalStrStiff(double [][] entirecoord,
          int [][] elemposition,
          double []ElasticMs, double []areas)
   {
      double [][] k=new
double[2*entirecoord.length][2*entirecoord.length];

      for(int i=0;i<k.length;i++)
      {
         for(int j=0;j<k[i].length;j++)
         {
            k[i][j]=0;
         }
      }
      for(int i=0;i<elemposition.length;i++)
      {
         double x1=entirecoord[elemposition[i][0]-1][0];
         double y1=entirecoord[elemposition[i][0]-1][1];
         double x2=entirecoord[elemposition[i][1]-1][0];
         double y2=entirecoord[elemposition[i][1]-1][1];
         double [][] xycoord={{x1,y1},{x2,y2}};
         double [][] littlek=Rod2Stiff(coord,ElasticMs[i],areas[i]);

                  int []ongfi=new int[4];
         for(int j=0;j<2;j++)
         {
            ongfi[j*2+0]=(elemposition[i][j])*2-1-1;
            ongfi[j*2+1]=(elemposition[i][j])*2-1;
         }
         for(int j=0;j<ongfi.length;j++)
         {
            for(int m=0;m<ongfi.length;m++)
            {
               k[ongfi[j]][ongfi[m]]+=littlek[j][m];
            }
         }
      }
      return k;
   }
```

Figure 3.5. PTGlobStrStiff module

3.2.2. Example Command for PTGlobStrStiff

The global structural stiffness matrix of the simple structure illustrated in Figure 3.6 can be evaluated by the following command.

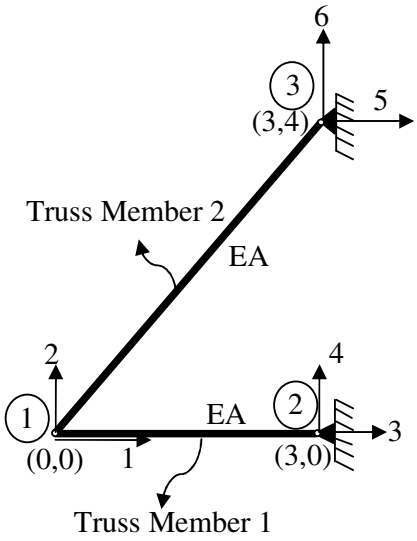


Figure 3.6. PTGlobStrStiff example

The command of PTGlobStrStiff [{(0,0),(3,0),(3,4)},{(1,2),(1,3)},{E,E},{A,A}]

calls: a truss plane system that is sketched in Figure 3.6 having
three joints with node coordinates (0,0),(3,0),(3,4),
two members which are interconnected with nodes 1-2 and 1-3,
two members with elastic modulus E,
two members with cross-section area A.

processes: local variable ongfi of each member
ongfi of member 1 = {1,2,3,4}
ongfi of member 2 = {1,2,5,6}

and returns: global structural stiffness matrix of the system given in equation (3.7).

$$K^G = AE\begin{bmatrix} 0.405 & 0.096 & -0.333 & 0 & -0.072 & -0.096 \\ 0.096 & 0.128 & 0 & 0 & -0.096 & -0.128 \\ -0.333 & 0 & 0.333 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -0.072 & -0.096 & 0 & 0 & 0.072 & 0.096 \\ -0,096 & -0.128 & 0 & 0 & 0.096 & 0.128 \end{bmatrix} \tag{3.7}$$

3.3. Implementation of Reduced Global Structural Stiffness System

As explained in the 2.3.5, displacement boundary conditions have to be applied in global structural stiffness matrix to prevent singularity.

The computer implementation of the reduced master stiffness is done by clearing appropriate rows and columns of the global structural stiffness matrix. Clearing of the restrained freedom indices of the system matrix is simply a rearrangement procedure. Rearrangement is carried out by assigning zero values to corresponding rows and columns that are affected by boundary conditions. Also related diagonal values are set to one. Applying, rearrangement to plane truss example, the reduced global structure stiffness becomes;

$$K^{G}_{reduced} = AE\begin{bmatrix} 0.405 & 0.096 & 0 & 0 & 0 & 0 \\ 0.096 & 0.128 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.8}$$

The Java code shown in Figure 3.7 called ReducedGlobStrStiff rearranges the global structural stiffness matrix with defined boundary conditions.

3.3.1. Formal Parameters of ReducedGlobStrStiff

Nboolean: In computer science, the Boolean data type, sometimes called the logical data type, is a primitive data type having two values: one and zero (which are equivalent to true and false). Nboolean is the node freedom information which contains boundary conditions. If the displacements are not restrained, related indices are set to False. On the other hand, if displacements are fixed by constraints, related indices are set to True.

K: Global structural stiffness matrix, "K^{G}", which reduction is needed.

Boundary conditions which are defined by Nboolean parameter are arranged with sub module called GlobalOrderPrescDisp. The purpose of this arrangement is to determine the global numbers of the prescribed degrees of freedom. Local variable globnopfi contains global order number of the prescribed freedom indices.

```
public static double[][] ReducedGlobStrStiff(boolean [][]Nboolean,
double [][]K)
   {  double [][]Kreduced=new double[K.length][];
      int n=K.length;
      LinkedList globnopfi=GlobalOrderPrescDisp(Nboolean);
      int np=globnopfi.size();
      for(int i=0;i<K.length;i++)
      {
         Kreduced[i]=new double[K[i].length];
         for(int j=0;j<K[i].length;j++)
            Kreduced[i][j]=K[i][j];
      }
      for(int k=0;k<np;k++)
      {
         int i=(Integer)(globnopfi.get(k));
         for(int j=0;j<n;j++)
         {Kreduced[i-1][j]=Kreduced[j][i-1]=0;}
         Kreduced[i-1][i-1]=1;
      }
      return Kreduced;
   }
   public static LinkedList GlobalOrderPrescDisp(boolean
[][]Nboolean)
   {
      LinkedList globnopfi=new LinkedList();
      int k=0;
      int numberofnodes=Nboolean.length;
      for(int n=1;n<=numberofnodes;n++)
      {
         int m=Nboolean[n-1].length;
         for(int j=1;j<=m;j++)
         {
            if(Nboolean[n-1][j-1])
               globnopfi.add(k+j);
         }
         k+=m;
      }
      return globnopfi;
   }
```

Figure 3.7. ReducedGlobStrStiff module

3.3.2. Example Command for ReducedGlobStrStiff

The reduced global structural stiffness matrix of the simple structure illustrated in Figure 3.6 can be evaluated by the following command.

The command of ReducedGlobStrStiff [{(0,0),(1,1),(1,1)},{ K^G }]

calls: global structural stiffness matrix K^G ,
with unrestrained joint in node 1
and with fixed supports in node 2 and 3

processes: local variable globnopfi
{3,4,5,6}

returns: reduced global structural stiffness matrix of the system which is given in equation (3.9).

$$K^G_{reduced} = AE\begin{bmatrix} 0.405 & 0.096 & 0 & 0 & 0 & 0 \\ 0.096 & 0.128 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.9}$$

3.4. Implementation of External Loads Considering Prescribed Displacements

Although the support conditions for plane trusses used in construction generally has zero initial displacement, there are cases where the initial displacement is nonzero. One of the well known initial displacement problem is the settlement of foundations. Figure 3.8 shows an initial displacement problem having a prescribed displacement at node 2 and an external load at node 1.

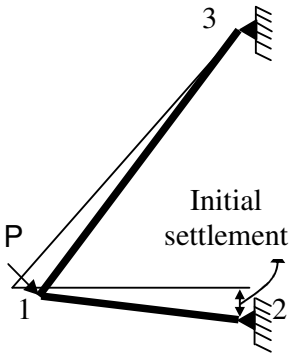


Figure 3.8. Initial settlement sample

Module in Figure 3.9 rearranges the node force matrix while considering the initial nonzero boundary conditions at supports. External loads which are defined by Nodevalues parameter are rearranged with sub module called ReducedNodeForces.

The purpose of this arrangement is to eliminate and recalculate the given node forces at joints while considering prescribed displacements.

3.4.1. Formal Parameters of ReducedNodeForces

Nboolean:	Nboolean is the node freedom information which contains boundary conditions. Freedom activity is labeled by True or False which corresponds 1 and 0, respectively. Parameter "True" means that the displacement is prescribed.
Nodevalues:	Initial values of the forces or prescribed displacements in the same order of Nboolean.
K:	Global Structural Stiffness matrix of the system
f:	External forces acting at each node

Local variable "vglobnopfi" is the values of the prescribed displacements listed in "globnopfi".

```
public static double [] ReducedNodeForces(boolean [][]Nboolean,
double [][]Nodevalues, double[][]K, double[]f)
   {
      int n=K.length;
      double []freduced=new double[f.length];
      for(int i=0;i<f.length;i++)
         freduced[i]=f[i];
      LinkedList globnopfi=KeCalculation.GlobalOrderPrescDisp
(Nboolean);
      int np=globnopfi.size();
      LinkedList
vglobnopfi=KeCalculation.ValuesPrescDisp(Nboolean,Nodevalues);
      int []c=new int[n];
      for(int i=0;i<n;i++)
         c[i]=1;
      for(int k=1;k<=np;k++)
      { int i=(Integer)(globnopfi.get(k-1));
         c[i-1]=0; }
      for(int k=1;k<=np;k++)
      {int i=(Integer)(globnopfi.get(k-1));
         double d=(Double)(vglobnopfi.get(k-1));
         freduced[i-1]=d;
         if(d==0)
            continue;
         for(int j=1;j<=n;j++)
            freduced[j-1]-=(K[i-1][j-1]*c[j-1]*d); }
      return freduced;
public static LinkedList ValuesPrescDisp(boolean
[][]Nboolean,double [][]Nvalues)
   {
      LinkedList vglobnopfi=new LinkedList();
      int k=0;
      int numberofnodes=Nboolean.length;
      for(int n=1;n<=numberofnodes;n++)
      {
         int m=Nboolean[n-1].length;
         for(int j=1;j<=m;j++)
         {
            if(Nboolean[n-1][j-1])
               vglobnopfi.add(Nvalues[n-1][j-1]);
         }
         k+=m;
      }
      return vglobnopfi;}
```

Figure 3.9. ReducedNodeForces module

3.4.2. Example Command for ReducedNodeForces

Considering P force at node 1 with P_{x1}=0 and P_{y1}=-10 load components, the following command evaluates the reduced node force matrix of the structure illustrated in Figure 2.12. The command of ReducedNodeForces[{(0,0),(1,1),(1,1)},{(0,-10),(0,0),(0,0)}, { K^G }, {(0,-10),(0,0),(0,0)}]

calls: the sketched structure system in Figure 2.12
with fixed supports in node 2 and 3
with external loads P_{y1}=-10
with [K^G] which is calculated in equation 3.7.

processes: local variable vglobnopfi and globnopfi

and returns: reduced node forces matrix of the structural system given in equation (3.10).

$$f_{reduced} = \begin{bmatrix} 0 \\ -10 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.10}$$

3.5. Implementation of Displacement Solutions

Boundary conditions and external loads of the structural system are implemented by modifying global stiffness matrix and global external force matrix, respectively. The reduced global structural stiffness matrix $[K_{reduced}]$ which contains force-displacement relationship for entire structure is used to solve all unknown displacements for any given loading scenario which were implemented in reduced node forces matrix [$f_{reduced}$]. Consequently, the linear system can be defined as;

$$[K_{reduced}][u_{unknown}] = [f_{reduced}] \tag{3.11}$$

The Java code shown in Figure 3.10 solves this linear system of equations.

3.5.1. Formal Parameters of JAMA.Matrix

Kreduced: The Reduced Global Structural Stiffness Matrix $[K_{reduced}]$

freduced: The Reduced Node Forces Matrix [$f_{reduced}$]

Jama.Matrix module solves the equation (3.11) and returns the unknown displacements of each node.

```
Jama.Matrix u=(new Jama.Matrix(Kreduced)).solve(new
Jama.Matrix(freduced,Kreduced.length));
double []unknwdisp=new double[Kreduced.length];
for(int i=0;i<unknwdisp.length;i++)
unknwdisp[i]=(u.getArray())[i][0];
for(int i=0;i<K.length;i++)
```

Figure 3.10. Jama matrix module

3.5.2. Example Command for JAMA.Matrix

The command of Jama.Matrix[Kreduced,freduced];

calls: the sketched structural system in Figure 2.12.
the reduced global stiffness matrix calculated in equation (3.9)
the reduced node forces matrix calculated in equation (3.10)

and returns: unknown displacements of the system that is defined in equation (3.11).

Substituting reduced stiffness matrix and reduced force matrix into equation (3.11), the linear system of equations becomes;

$$AE\begin{bmatrix} 0.405 & 0.096 & 0 & 0 & 0 & 0 \\ 0.096 & 0.128 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} u_{unknown1} \\ u_{unknown2} \\ u_{unknown3} \\ u_{unknown4} \\ u_{unknown5} \\ u_{unknown6} \end{bmatrix} = \begin{bmatrix} 0 \\ -10 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.12}$$

And Jama.Matrix module returns unknown displacement solutions as follows:

$$\begin{bmatrix} u_{unknown1} \\ u_{unknown2} \\ u_{unknown3} \\ u_{unknown4} \\ u_{unknown5} \\ u_{unknown6} \end{bmatrix} = \begin{bmatrix} 22.5/EA \\ -95/EA \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{3.13}$$

3.6. Implementation of Node Reaction Force Solutions

In order to calculate node forces, the unknown displacements that are solved by Jama.Matrix Module are used. Since the global structural stiffness equation is given as;

$$f^G = K^G u_{solved} \tag{3.14}$$

The node forces including reactions are obtained by multiplying global structural stiffness matrix with the displacement matrix. Java code expressed in equation (3.15) calculates the node forces.

```
f[i]=MatrixOperations.dot_product(K[i],unknwdisp);
```
(3.15)

3.6.1. Example Calculation of Node Reaction Forces

The node forces including reactions of the sample structure sketched in Figure 2.13 can be find by multiplying global structural matrix and the solution of displacement matrix

which is obtained in equation (3.13). Substituting displacement solution matrix and K^G into equation (3.14), the node forces can be found as follows:

$$\begin{bmatrix} f_{x1} \\ f_{y1} \\ f_{x2} \\ f_{y2} \\ f_{x3} \\ f_{y3} \end{bmatrix} = \begin{bmatrix} K^G \end{bmatrix} \begin{bmatrix} +22.5/EA \\ -95/EA \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ -10 \\ -7.5 \\ 0 \\ 7.5 \\ 10 \end{bmatrix} \tag{3.16}$$

Same reaction forces were found in equation (2.79) and (2.81) and solutions were sketched in Figure 2.14.

3.7. Implementation of Internal Force and Stress Solutions

Rod2InternatFrc module in Figure 3.11, computes the internal forces of any rod element by using the equation (2.74).

```
public static double Rod2InternalFrc(double [][]Coordinates, double
ElasticM, double area, double []udisp)
   {double EA=ElasticM*area;
      double x21=Coordinates[1][0]-Coordinates[0][0];
      double y21=Coordinates[1][1]-Coordinates[0][1];
      double LL=x21*x21+y21*y21;
      double axialfe=(EA/LL)*(x21*(udisp[2]-
udisp[0])+y21*(udisp[3]-udisp[1]));
      return axialfe;}
```

Figure 3.11. Rod2InternatFrc module

3.7.1. Formal Parameters of Rod2InternalFrc

Coordinates: Global coordinates of the nodes of the rod element {(x1,y1),(x2,y2)}

ElasticM: Elastic modulus of the rod element (E)

area: Cross-section area of the rod element (A)

udisp: Displacements solved by Jama.Matrix

Finally, average axial stress σ is obtained by dividing axial forces by the cross-sectional area of the member. Therefore, the axial stresses of the rod element are implemented by Rod2Strs module which is shown in Figure 3.12.

```
public static double [] Rod2Strs(double []area, double
[]elementaxialf)
{
int numberofele=area.length;
double []elementstress=new double[numberofele];
for(int i=0;i<numberofele;i++)
elesig[i]=0;
for(int e=1;e<=numberofele;e++)
elementstress[e-1]=elementaxialf[e-1]/area[e-1];
return elementstress;
}
```

Figure 3.12. Rod2Strs module

3.7.2. Formal Parameters of Rod2Strs

area: Cross-section area of the rod element (A)

***elementaxialf*:** Element axial forces which are calculated in Rod2InternatFrc module.

The algorithms of the master stiffness method introduced in this section will be used for the development of the further Java programs.

4. JAVA PROGRAM FOR 3D TRUSSES

4.1. Introduction

Three dimensional trusses are very efficient structures to carry heavy loads as well as span long distances. A simple three dimensional plane truss is depicted in Figure 4.1. Since the fundamental theory for three dimensional trusses has existed for over a century; such structural systems have been employed for different engineering applications from the beginning of 20^{th} century. However, in the past, due to the large number of degrees of freedom in the analysis of three dimensional trusses was a challenging issue. The growth of computational power together with software developments have the engineers had the ability to solve large three-dimensional systems by finite element programs.

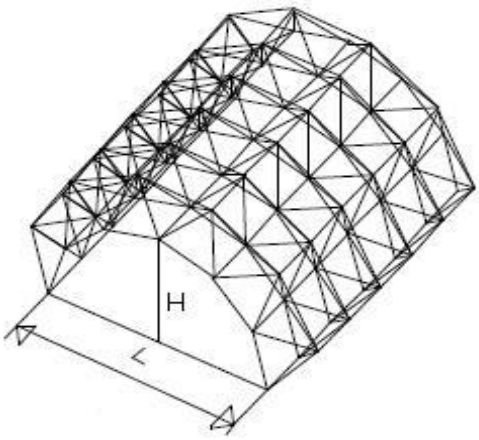


Figure 4.1. 3D Truss example

The development of the three dimensional finite element program is a simple extension of the matrix stiffness method presented for the one-dimensional structural systems that is introduced previously.

A three dimensional truss structure analysis program with Java language is consists of four major stages which are valid for the whole structural analysis programs used today such as Sap2000 and Etabs. The analysis stages involved can be summarized as; definition, execution, solution and computation.

Definition stage is simply an input preparing process to define the problem. Input file contains geometry and element data of the model. Furthermore, boundary conditions and external loads are introduced by input file. Execution stage is the implementation of the matrix stiffness method for the given input data. In the first step of execution, local element modules that are introduced are organized to develop global structural stiffness matrix. Than, boundary conditions and node force data are used to modify global structural stiffness matrix to calculate reduced global structural stiffness matrix. In the solution stage, the unknown displacement of the given system is found by using Java Linear solver. Finally, in computation stage, recovery forces including reactions, internal forces and stresses in each truss member are calculated. The results obtained from driver program are written in output file. For three dimensional truss problems, it is required to define a three dimensional rod element which has 3 degrees of freedom at each node.

4.2. Implementation of 3D Rod Element

The 3D rod element which is depicted in Figure 4.2 has a modulus of elasticity of "E" and constant cross section area of "A".

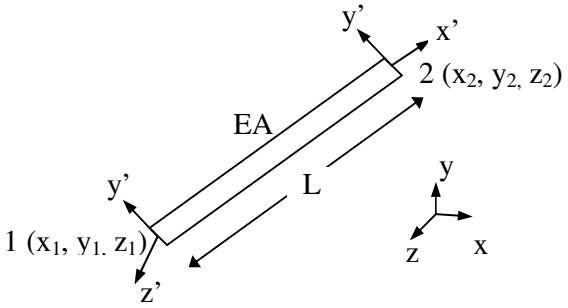


Figure 4.2. 3D Rod element

Node displacements and corresponding node forces of the rod element are defined as

$$u' = \begin{bmatrix} u'_{x1} \\ u'_{y1} \\ u'_{z1} \\ u'_{x2} \\ u'_{y2} \\ u'_{z2} \end{bmatrix}, f' = \begin{bmatrix} f'_{x1} \\ f'_{y1} \\ f'_{z1} \\ f'_{x2} \\ f'_{y2} \\ f'_{z2} \end{bmatrix} \tag{4.1}$$

The implemented Rod2stiff module in Figure 3.3 can be improved to three dimensions by using equation 3.2. The global stiffness Java module of three dimensional rod element is shown in Figure 4.3.

```
public static double[][] a3DRod3Stiff(double [][]XYZCoord,
double ElasticM, double area)
double x21=XYZCoord[1][0]-XYZCoord[0][0];
double y21=XYZCoord[1][1]-XYZCoord[0][1];
double z21=XYZCoord[1][2]-XYZCoord[0][2];
double
factor=ElasticM*area/(Math.pow(Math.sqrt(x21*x21+y21*y21
+z21*z21),3));
double [][] ke=
{{ x21*x21, x21*y21, x21*z21,-x21*x21,-x21*y21,-x21*z21},
{ y21*x21, y21*y21, y21*z21,-y21*x21,-y21*y21,-y21*z21},
{ z21*x21, z21*y21, z21*z21,-z21*x21,-z21*y21,-z21*z21},
{-x21*x21,-x21*y21,-x21*z21, x21*x21, x21*y21, x21*z21},
{-y21*x21,-y21*y21,-y21*z21, y21*x21, y21*y21, y21*z21},
{-z21*x21,-z21*y21,-z21*z21, z21*x21, z21*y21, z21*z21}};
ke=MatrixOperations.multiply_with_scalar(ke,factor);
return ke;  }
```

Figure 4.3. a3DRod3Stiff module

4.2.1. Formal Parameters of a3DRod3Stiff:

XYZCoord: Global coordinates of the nodes of the rod element {(x1, y1, z1), (x2, y2, z2)}

ElasticM: Elastic modulus of the rod elements (E)

area: Cross-section area of the rod elements (A)

The a3DRod3Stiff module returns the 6x6 global element stiffness matrix of the three dimensional rod element.

4.3. Implementation of Global Structural Stiffness Matrix

The Java code shown in Figure 4.4 called a3DTrussGlobStrStiff returns global structural matrix of three dimensional truss systems.

Global structural stiffness matrix, "K^G" is a square matrix which accommodates all displacement force relations of every degree of freedoms. Since plane truss elements have 3 freedoms in every node, the total number of the freedom is equal to;

$$\Sigma n_{freedom} = \text{total number of freedom} = 3^* \text{ number of nodes} \quad (4.2)$$

Thus, global structural stiffness matrix of three dimensional structures has ($\Sigma n_{freedom}$* $\Sigma n_{freedom}$) dimensions.

```
public static double [][] a3DTrussGlobStrStiff(double
[][]allXYZCoord, int [][]elempositions,double []ElasticMs,  double
[]areas)
   {
      int numberofele=elempositions.length;
      int numberofnodes=allXYZCoord.length;
      double
[][]K=MatrixOperations.zero_matrix(3*numberofnodes);
      for(int e=1;e<=numberofele;e++)
      {
         int ni=elempositions[e-1][0];
         int nj=elempositions[e-1][1];
         int []ongfitab={3*ni-2,3*ni-1,3*ni,3*nj-2,3*nj-1,3*nj};
         double [][]XYCoord={allXYZCoord[ni-1],allXYZCoord[nj-
1]};
         double Em=ElasticMs[e-1];
         double A=areas[e-1];
         double [][]Ke=a3DRod3Stiff (XYCoord,Em,A);
         for(int i=1;i<=Ke.length;i++)
         {
            int ii=ongfitab[i-1];
            for(int j=i;j<=Ke.length;j++)
            {
               int jj=ongfitab[j-1];
               K[ii-1][jj-1]+=Ke[i-1][j-1];
               K[jj-1][ii-1]=K[ii-1][jj-1];
            }
         }
      }
      return K;
   }
```

Figure 4.4. a3DTrussGlobStrStiff module

4.3.1. Formal Parameters of a3DTrussGlobStrStiff

allXYZcoord: All global coordinates of the system joints
{(x1, y1, z1), (x2, y2, z2)…….(x_{last}, y_{last})}

elempositions: Element end points that are assigned for the system joints which are required to define an element and its local axis.
$\{(\#1, \#2)_1, (\#1, \#2)_2 \ldots\ldots (\#1, \#2)_{last}\}$

ElasticMs: Elastic Modulus of each element in the same order of "elemposition" parameter
(E_1,E_2….E_{last})

areas: Cross-sections of each element in the same order of "elemposition" parameter
(A_1,A_2….A_{last})

a3DTrussGlobStrStiff module returns $\Sigma n_{freedom} * \Sigma n_{freedom}$ global structural stiffness matrix of the 3D truss structures.

4.4. Implementation of Reduced Global Structural Stiffness System

Displacement boundary conditions of the three dimensional truss systems have to be applied in global structural stiffness matrix to prevent singularity. The computer implementation of the boundary conditions is carried out by the ReducedGlobStrStiff module which is given in Figure 3.7. The formal parameters of the ReducedGlobStrStiff module are repeated for convenience.

Nboolean: Freedom information which contains boundary conditions

K: Global structural stiffness matrix, "K^G", which reduction is needed.

ReducedGlobStrStiff module returns $\Sigma n_{freedom} * \Sigma n_{freedom}$ reduced global structural stiffness matrix of the 3D truss structures.

4.5. Implementation of External Loads Considering Prescribed Displacements

ReducedNodeForces Module in Figure 3.9 rearranges the node force matrix while considering the initial nonzero boundary conditions at supports.

The purpose of this arrangement is to eliminate and recalculate the given node forces at joints while considering prescribed displacements. The formal parameters of the ReducedNodeForces module are repeated for convenience.

Nboolean:	Freedom information which contains boundary conditions
Nodevalues:	Initial values of the forces or prescribed displacements in the same order of Nboolean.
K:	Global Structural Stiffness matrix of the system
f:	External forces acting at each node

ReducedGlobStrStiff module returns $\Sigma n_{freedom}$ x 1 reduced node forces matrix of the 3D truss structures.

4.6. Implementation of Displacement Solutions

The Reduced global structural stiffness matrix $[Kreduced]$ which contains force-displacement relationship for entire structure is used to solve all unknown displacements for any given loading scenario which were implemented in reduced node forces matrix $[freduced]$.

A3DTrussSolution module shown in Figure 4.5 solves the linear system of equations of the three dimensional truss systems by using equation (3.11).

```
  public static Object [] A3DTrussSolution(double
[][]allXYZCoord,
int [][]elempositions, double []ElasticMs, double []areas, boolean
[][]NBoolean, double [][]NodeValues)
 {
      double
[][]K=KeCalculation.A3DTrussGlobStrStiff(allXYZCoord,elempos
itions,ElasticMs,areas);

      double [][]Kreduced=KeCalculation. ReducedGlobStrStiff
(NBoolean,K);

      double []f=new
double[NodeValues.length*NodeValues[0].length];
      for(int i=0,k=0;i<NodeValues.length;i++)
      {
         for(int j=0;j<NodeValues[i].length;j++)
         {
            f[k]=NodeValues[i][j];
            k++;
         }
      }
      double
[]freduced=NodeForces.ReducedNodeForces(NBoolean,NodeValue
s,K,f);

      Jama.Matrix u=(new Jama.Matrix(Kreduced)).solve(new
Jama.Matrix(freduced,Kreduced.length));
      double []unknwndisp=new double[Kreduced.length];
      for(int i=0;i<unknwndisp.length;i++)
         unknwndisp[i]=(u.getArray())[i][0];
      for(int i=0;i<K.length;i++)

      {
         f[i]=MatrixOperations.dot_product(K[i],unknwndisp);
      }
```

Figure 4.5. a3DTrussSolution module

4.6.1. Formal Parameters of a3DTrussSolution

allXYZcoord: All global coordinates of the system joints
{(x1, y1, z1), (x2, y2, z2)…….(x_{last}, y_{last})}

elempositions: Element end points that are assigned for the system joints which are required to define an element and its local axis.

$\{(\#1, \#2)_1, (\#1, \#2)_2 \ldots\ldots (\#1, \#2)_{last}\}$

ElasticMs: Elastic Modulus of each element in the same order of "elemposition" parameter
$(E_1, E_2 \ldots E_{last})$

areas: Cross-sections of each element in the same order of "elemposition" parameter
$(A_1, A_2 \ldots A_{last})$

Nboolean: Freedom information which contains boundary conditions

Nodevalues: Initial values of the forces or prescribed displacements in the same order of Nboolean.

a3DTrussSolution module returns the unknown displacements of each node.

4.7. Implementation of Internal Force and Stress Solutions

Module a3DTrussInternalFrc given in Figure 4.6, computes the internal forces of three dimensional rod elements by using the equation (2.80). Besides, member stresses are implemented by a3DTrussStrs module which is shown in Figure 4.7.

4.7.1. Formal Parameters of a3DTrussInternalFrc

allXYZcoord: All global coordinates of the system joints
$\{(x1, y1, z1), (x2, y2, z2) \ldots\ldots (x_{last}, y_{last})\}$

elempositions: Element end points that are assigned for the system joints which are required to define an element and its local axis.
$\{(\#1, \#2)_1, (\#1, \#2)_2 \ldots\ldots (\#1, \#2)_{last}\}$

ElasticMs: Elastic Modulus of each element in the same order of "elemposition" parameter $(E_1, E_2 \ldots E_{last})$

areas: Cross-sections of each element in the same order of "elemposition" parameter
$(A_1, A_2 \ldots A_{last})$

dissolved: Displacements of the nodes solved by Jama Matrix module
$(u_{x1}, u_{x2} \ldots u_{xlast})$

a3DTrussInternalFrc module returns the axial forces of each element.

```
Public static double [] a3DtrussInternalFrc
(double [][]allXYZcoord, int [][]elempositions,
double []ElasticMs, double []areas, double [][]dissolved)
   {
      int numberofnodes=allXYZcoord.length;
      int numberofele=elempositions.length;
      double []p=new double[numberofele];
      for(int i=0;i<p.length;i++)
         p[i]=0;
      for(int e=1;e<=numberofele;e++)
      {
         int ni=elempositions[e-1][0];
         int nj=elempositions[e-1][1];
         double ijcoord[][]={allXYZcoord[ni-1],allXYZcoord[nj-1]};
         double []ue=new double[dissolved[ni-
1].length+dissolved[nj-1].length];
         for(int i=0;i<dissolved[ni-1].length;i++)
            elemdisp[i]=dissolved[ni-1][i];
         for(int i=0;i<dissolved[nj-1].length;i++)
            elemdisp[dissolved[ni-1].length+i]=dissolved[nj-1][i];
         p[e-1]= a3Drod3Stiff IntForce(ijcoord,ElasticMs[e-
1],areas[e-1],elemdisp);
      }
      return p;
   }
   public static double a3Drod3StiffInternalFrc(double [][]ijcoord,
double Em, double A, double []elemdisp)
   {
      double x21=ijcoord[1][0]-ijcoord[0][0];
      double y21=ijcoord[1][1]-ijcoord[0][1];
      double z21=ijcoord[1][2]-ijcoord[0][2];
      double EA=Em*A;
      double LL=x21*x21+y21*y21+z21*z21;
      double pe=(EA/LL)*(x21*(elemdisp[3]-
elemdisp[0])+y21*(elemdisp[4]-elemdisp[1])+z21*(elemdisp[5]-
elemdisp[2]));
      return pe;
   }
}
```

Figure 4.6. a3DTrussInternalFrc module

4.7.2. Formal Parameters of a3DTrussStrs

areas: Cross-sections of each element in the same order of "elemposition" parameter (A_1,A_2….A_{last})

ElementForces: Element axial forces which are evaluated by a3DtrussInternalFrc module. (F_1,F_2….F_{last})

3DtrussStr module returns the element stresses of the three dimensional truss structure.

```
Public static double [] a3DtrussStrs(double []areas, double
[]ElementForces)
   {
      int numberofele=areas.length;
      double []elementstress=new double[numberofele];
      for(int i=0;i<numberofele;i++)
         elementstress[i]=0;
      for(int e=1;e<=numberofele;e++)
         elementstress[e-1]=ElementForces[e-1]/areas[e-1];
      return elementstress;
   }
```

Figure 4.7. a3DTrussStrs module

4.8. Implementation of Results

Java object called Result which is shown in Figure 4.7 collects the solutions of three dimensional truss problems. Results of the system are gathered with the four local parameters and transmitted to the Output Handler.

Local parameter "dissolved" collects the displacement solution matrix which is evaluated by Jama.Matrix module. Local parameter "NForces" collects reaction forces of the system by applying equation (3.14). Local parameter "Eintforces" collects internal forces of the elements by using equation (2.80).

Finally, local parameter "estresses" collects the element stresses evaluated by a3DTrussStrs module.

```
Jama.Matrix u=(new Jama.Matrix(Kmod)).solve(new
Jama.Matrix(fmod,Kmod.length));
      double []my_u=new double[Kmod.length];
      for(int i=0;i<my_u.length;i++)
         my_u[i]=(u.getArray())[i][0];

      for(int i=0;i<K.length;i++)
      {
         f[i]=MatrixOperations.dot_product(K[i],my_u);
      }

      double
[]elementaxialf=NodeForces.a3DTrussInternalFrc(allXYZcoord,
elempositions,ElasticMs,areas,dissolved);
      double
[]elementstress=StressSolutions.a3DTrussStrs(areas,elementaxialf);

Object []result=new Object[4];
      result[0]=dissolved;
      result[1]=Nforces;
      result[2]=Eintforces;
      result[3]=estresses;
      return result;
```

Figure 4.8. Result object for 3D Truss structures

4.9. Flow Chart of the 3D Truss Program

Schematic representation of the algorithm of the three dimensional truss program is given in Figure 4.9.

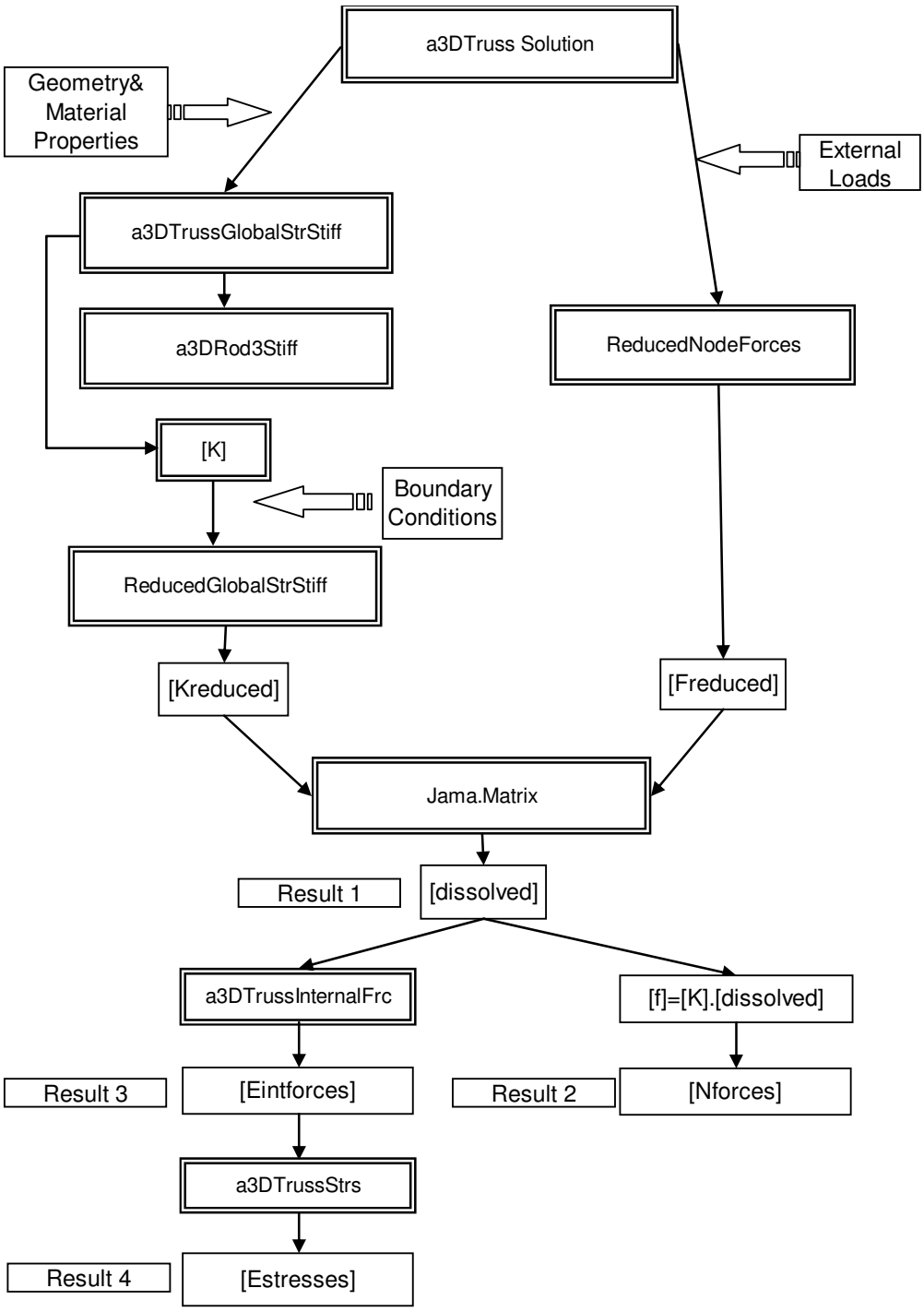


Figure 4.9. Flowchart of the 3D Truss program

4.10. Case Study of a Plane Truss Problem

The purpose of this section is to compare the Java three dimensional truss program with the SAP2000. A simple plane truss problem which is shown in Figure 4.10 is used to test the algorithm. The plane truss structure has 13 members and 7 nodes with 3 joint loads applying at the sketched points. The properties of the problem are given in the Table 4.1. Input file called 1space_truss_input.xls which is prepared by Microsoft Excel program is given in the Table 4.2 and in Appendix B.

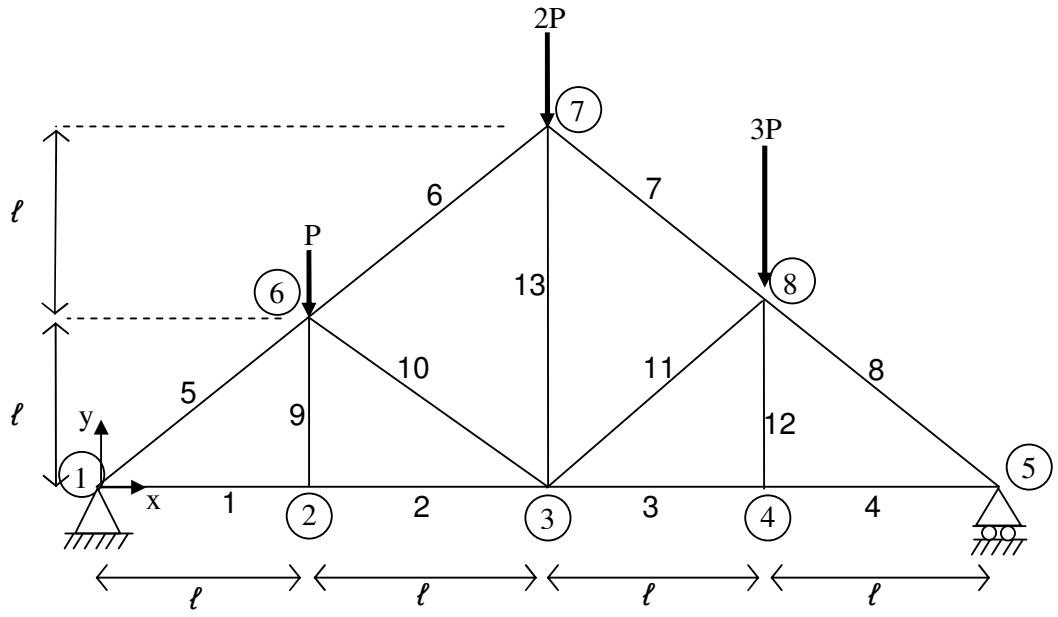


Figure 4.10. Plane truss problem

Table 4.1. Properties of the plane truss problem

P	1 kN
ℓ	3 m
Elastic Modulus (E)	$21 \cdot 10^6$ kN / m^2
Cross-section of beams	0.05 m x 0.05 m

Table 4.2. Input data of the plane truss problem

Ref No	Coordinates			Element End Nodes		Elastic Modulus (kN/m^2)	Areas (m^2)	Node Constraints			Joint Loads		
	x	y	z	i	j			x	y	z	x	y	z
1	0	0	0	1	2	21000000	0.0025	1	1	1	0	0	0
2	3	0	0	2	3	21000000	0.0025	0	0	1	0	0	0
3	6	0	0	3	4	21000000	0.0025	0	0	1	0	0	0
4	9	0	0	4	5	21000000	0.0025	0	0	1	0	0	0
5	12	0	0	1	6	21000000	0.0025	0	1	1	0	0	0
6	3	3	0	6	7	21000000	0.0025	0	0	1	0	-5	0
7	6	6	0	7	8	21000000	0.0025	0	0	1	0	-10	0
8	9	3	0	8	5	21000000	0.0025	0	0	1	0	-15	0
9				2	6	21000000	0.0025						
10				6	3	21000000	0.0025						
11				3	8	21000000	0.0025						
12				4	8	21000000	0.0025						
13				3	7	21000000	0.0025						

Member element stresses and the displacements of the nodes of the given plane truss problem are calculated by Java 3D truss program and Sap2000. The results are compared in Table 4.3 and Table 4.4.

Table 4.3. Comparison of element stresses

Elements	First Joint	Last Joint	Stress (kN)	
			Java	Sap2000
1	1	2	12.5	12.5
2	2	3	12.5	12.5
3	3	4	17.5	17.5
4	4	5	17.5	17.5
5	1	6	-17.6776	-17.6777
6	6	7	-14.1421	-14.1421
7	7	8	-14.1421	-14.1421
8	8	5	-24.7487	-24.7487
9	2	6	0	0
10	6	3	-3.5355	-3.535
11	3	8	-10.6066	-10.6066
12	4	8	0	0
13	3	7	10	10

Table 4.4. Comparison of displacements

Joint	Displacements (m)(10^{-4})					
	Java		Sap2000		Comparison (%)	
	x	y	x	y	x	y
1	0	0	0	0	100	100
2	7.1429	-53.7535	7.14	-53.75	100	100
3	14.2857	-68.9775	14.29	-68.98	100	100
4	24.2857	-64.6918	24.29	-64.69	100	100
5	34.2857	0	34.29	0	100	100
6	33.5504	-53.7535	33.55	-53.75	100	100
7	21.1835	-57.549	21.18	-57.55	100	100
8	-2.1218	-64.6918	-2.12	-64.69	100	100

It is shown that Java three dimensional truss program evaluates the same results for the given plane truss problem with Sap2000.

4.11. Case Study of a 3D Truss Problem

In order to compare the Java program with a three dimensional truss structure the problem illustrated in Figure 4.11 is used.

The three dimensional two-storey truss structure has 24 members and 12 joint nodes. Three P forces are applied to the node 3. The properties of the problem are given in the Table 4.5. Input file called 2plane_truss_input.xls which is prepared by Microsoft Excel program is given in the Table 4.6 and in Appendix B.

Table 4.5. Properties of the 3D Truss problem

P	10 kN
ℓ	3 m
Elastic Modulus (E)	$21 . 10^6$ kN / m^2
Cross-section of beams	0.05 m x 0.05 m

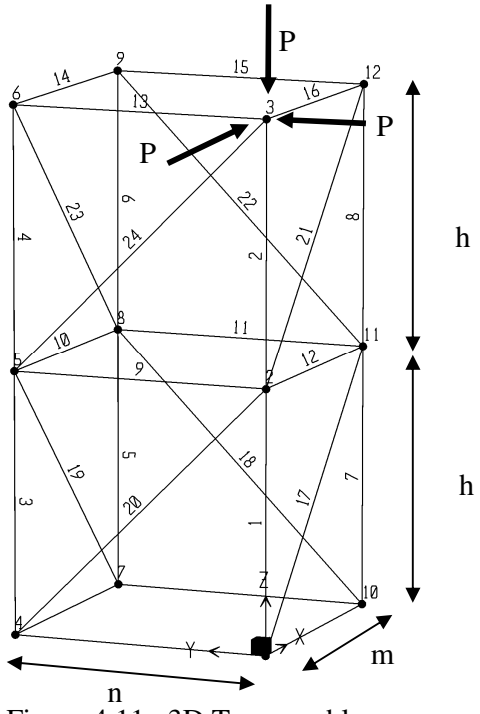


Figure 4.11. 3D Truss problem

Table 4.6. Input data of the 3D Truss problem

Ref No	Coordinates			Element End Nodes		Elastic Modulus	Areas	Node Constraints			Joint Loads		
	x	y	z	i	j			x	y	z	x	y	z
1	0	0	0	1	2	21000000	0.0025	1	1	1	0	0	0
2	0	0	5	2	3	21000000	0.0025	0	0	0	0	0	0
3	0	0	10	4	5	21000000	0.0025	0	0	0	10	10	-10
4	0	5	0	5	6	21000000	0.0025	1	1	1	0	0	0
5	0	5	5	7	8	21000000	0.0025	0	0	0	0	0	0
6	0	5	10	8	9	21000000	0.0025	0	0	0	0	0	0
7	5	5	0	10	11	21000000	0.0025	1	1	1	0	0	0
8	5	5	5	11	12	21000000	0.0025	0	0	0	0	0	0
9	5	5	10	2	5	21000000	0.0025	0	0	0	0	0	0
10	5	0	0	5	8	21000000	0.0025	1	1	1	0	0	0
11	5	0	5	8	11	21000000	0.0025	0	0	0	0	0	0
12	5	0	10	11	2	21000000	0.0025	0	0	0	0	0	0
13				3	6	21000000	0.0025						
14				6	9	21000000	0.0025						
15				9	12	21000000	0.0025						
16				12	3	21000000	0.0025						
17				1	11	21000000	0.0025						
18				10	8	21000000	0.0025						
19				7	5	21000000	0.0025						
20				4	2	21000000	0.0025						
21				2	12	21000000	0.0025						
22				11	9	21000000	0.0025						
23				8	6	21000000	0.0025						
24				5	3	21000000	0.0025						

Member element stresses and the displacements of the nodes of the given 3D truss problem are calculated by Java 3D truss program and Sap2000. The results are compared in Table 4.7 and Table 4.8.

Table 4.7. Comparison of element stresses

Beams	First Joint	Last Joint	Stress (kN)		
			Java	Sap2000	Comparison (%)
1	1	2	20	20	100
2	2	3	0	0	100
3	4	5	-10	-10	100
4	5	6	0	0	100
5	7	8	0	0	100
6	8	9	0	0	100
7	10	11	-20	-20	100
8	11	12	-10	-10	100
9	2	5	10	10	100
10	5	8	0	0	100
11	8	11	0	0	100
12	11	2	-10	-10	100
13	3	6	0	0	100
14	6	9	0	0	100
15	9	12	0	0	100
16	12	3	-10	-10	100
17	1	11	14.142	14.142	100
18	10	8	0	0	100
19	7	5	0	0	100
20	4	2	-14.142	-14.142	100
21	2	12	14.142	14.142	100
22	11	9	0	0	100
23	8	6	0	0	100
24	5	3	-14.142	-14.142	100

Table 4.8. Comparison of displacements

Joints	Displacements (m)(10^{-4})								
	Java			SAP2000			Comparison (%)		
	X	y	z	x	y	z	x	y	z
1	0	0	0	0	0	0	100	100	100
2	55.51	45.99	19.05	55.51	45.99	19.05	100	99.99	99.99
3	139.59	111.02	19.05	139.59	111.02	19.05	100	100	99.99
4	0	0	0	0	0	0	100	100	100
5	-9.52	55.51	-9.52	-9.52	55.51	-9.52	99.96	100	99.96
6	-19.05	111.02	-9.52	-19.05	111.02	-9.52	99.99	100	99.96
7	0	0	0	0	0	0	100	100	100
8	-9.52	0	0	-9.52	0	0	99.96	100	100
9	-19.05	-19.05	0	-19.05	-19.05	0	99.99	99.99	100
10	0	0	0	0	0	0	100	100	100
11	45.99	0	-19.05	45.99	0	-19.05	99.99	100	99.99
12	130.07	-19.05	-28.58	130.07	-19.05	-28.57	100	99.99	100

It is shown that Java three dimensional truss program evaluates the same results for the given 3D truss problem with Sap2000.

5. JAVA PROGRAM FOR BEAM-COLUMN FRAME STRUCTURES

5.1. Introduction

Beam-column frames are structural assemblies, of beam and beam-column elements that transfer loads mainly by flexure. The elements are connected by rigid or pinned joints and the loads are applied in the plane of the frame. The frame members usually define spaces into bays and storeys, and frame structures are classified into different types based on the number of bays and storeys they create. Two examples of portal frames are depicted in Figure 5.1.

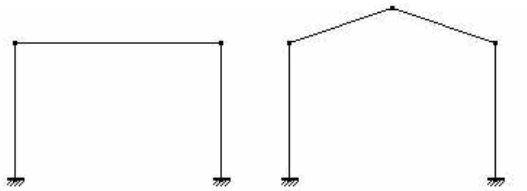


Figure 5.1. Plane frame examples

A typical member in a rigid frame is subjected to axial force, shear force and bending moment at any section along its length. These are referred to as internal forces and arise to resist the external loads applied to the structure. Figure 5.2 shows the internal forces induced in the elements meeting at a rigid joint by considering free body diagrams of beam, column and rigid joint elements.

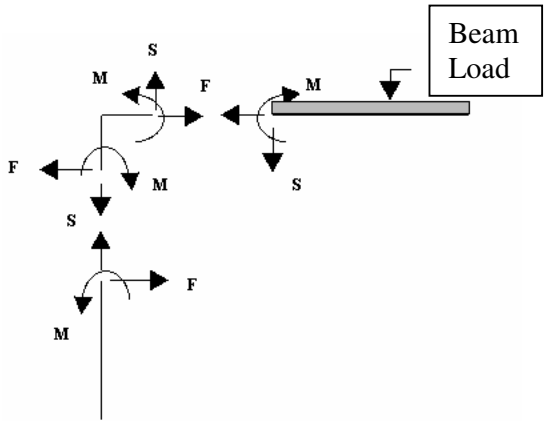


Figure 5.2. Internal forces of a beam

The four major analysis stages, “definition, execution, solution and computation”, introduced in 4.1 are also valid for Beam-Column Frame programs. For beam-column frame problems, it is required to define a beam element which has three degrees of freedom at each node.

5.2. Implementation of Beam Element Stiffness Matrix

Since the members of the skeletal structures required to be resist axial, shear and bending actions, beam element with six degrees of freedom is needed for the solution of two beam-column frame problems.

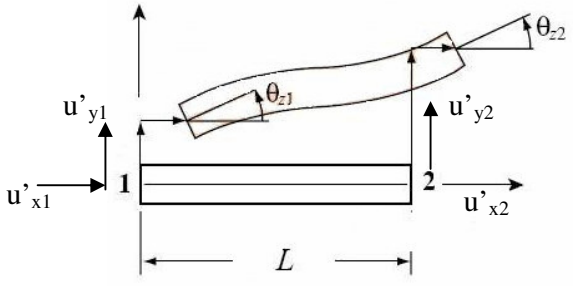


Figure 5.3. Beam element

Beam element shown in Figure 5.3 has three degrees of freedom at each node. The length of the element is L and has a constant elastic module of E. The moment of inertia is taken to be constant about the neutral axis and equals to I_{zz}. Node displacements and corresponding node forces of the beam element are defined as

$$u' = \begin{bmatrix} u'_{x1} \\ u'_{y1} \\ \theta'_{z1} \\ u'_{x2} \\ u'_{y2} \\ \theta'_{z2} \end{bmatrix}, f' = \begin{bmatrix} f'_{x1} \\ f'_{y1} \\ m'_{z1} \\ f'_{x2} \\ f'_{y2} \\ m'_{z2} \end{bmatrix} \tag{5.1}$$

Local stiffness matrix of a simple beam element with four degrees of freedom derived in equation (2.31) and local stiffness matrix of a rod element with four degrees of freedom derived in equation (2.48) are repeated here for convenience:

$$\frac{EI_{zz}}{L^3}\begin{bmatrix} 12 & 6L & -12 & 6L \\ 6L & 4L^2 & -6L & 2L^2 \\ -12 & -6L & 12 & -6L \\ 6L & 2L^2 & -6L & 4L^2 \end{bmatrix}\begin{bmatrix} y_1 \\ \theta_1 \\ y_2 \\ \theta_2 \end{bmatrix} = \begin{bmatrix} f_{z1} \\ m_1 \\ f_{z2} \\ m_2 \end{bmatrix} \tag{5.2}$$

$$\frac{EA}{L}\begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} u'_{x1} \\ u'_{y1} \\ u'_{x2} \\ u'_{y2} \end{bmatrix} = \begin{bmatrix} f'_{x1} \\ f'_{y1} \\ f'_{x2} \\ f'_{y2} \end{bmatrix} \tag{5.3}$$

The local stiffness of the beam element sketched in figure 5.3 is obtained by adding the stiffness matrices indicated in equations (5.2) and (5.3).

The local stiffness is calculated after rearranging rows and columns in accordance with the nodal freedoms in equation (5.1) as follows;

$$[K'] = \frac{EA}{L}\begin{bmatrix} 1 & 0 & 0 & -1 & 0 & 0 \\ & 0 & 0 & 0 & 0 & 0 \\ S & & 0 & 0 & 0 & 0 \\ & Y & . & 1 & 0 & 0 \\ & . & M & & 0 & 0 \\ . & & & M & & 0 \end{bmatrix} + \frac{EI_{zz}}{L^3}\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ & 12 & 6L & 0 & -12 & 6L \\ S & & 4L^2 & 0 & -6L & 2L^2 \\ & Y & . & 0 & 0 & 0 \\ & . & M & & 12 & -6L \\ . & & & M & & 4L^2 \end{bmatrix} \tag{5.4}$$

In order to obtain the global element stiffness matrix, transformation equation derived in equation (2.61) is required and it is repeated in equation (5.5).

$$k^e = T^T k' T \tag{5.5}$$

Since the rotation angles θ and nodal moments *m* are same in both local and global coordinates, transformation matrix can be expanded from equation (2.51) as follows

$$[T] = \begin{bmatrix} \cos\alpha & \sin\alpha & 0 & 0 & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & 0 & 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.6}$$

5.2.1. Formal Parameters of Beam3Stiff:

XYCoord: Global coordinates of the nodes of the rod element {(x1, y1), (x2, y2)}

ElasticM: Elastic modulus of the rod element (E)

area: Cross-section area of the rod element (A)

InertiaZ: Moment of inertia about the neutral axis (I_{zz})

Beam3Stiff module given in Figure 5.4, executes required matrix multiplications and returns 6x6 global stiffness matrix of the beam element.

```
 public static double[][] Beam3Stiff(double [][]XYCoord, double
ElasticM, double area, double InertiaZ)
   {
      double x21=XYCoord[1][0]-XYCoord[0][0];
      double y21=XYCoord[1][1]-XYCoord[0][1];
      double LL=x21*x21+y21*y21;
      double L=Math.sqrt(LL);
      double LLL=LL*L;
      double constant1=ElasticM*area/L;//ElasticM*A/LLL
      double constant2=2*ElasticM*InertiaZ/LLL;
      double [][] krod={
         { 1,0,0,-1,0,0},{0,0,0,0,0,0},{0,0,0,0,0,0},
         {-1,0,0, 1,0,0},{0,0,0,0,0,0},{0,0,0,0,0,0}};
      double [][] kbeam={
         { 0,0,0,0,0,0},{0, 6, 3*L,0,-6, 3*L},{0,3*L,2*LL,0,-3*L,
LL},
         { 0,0,0,0,0,0},{0,-6,-3*L,0, 6,-3*L},{0,3*L, LL,0,-
3*L,2*LL}};

krod=MatrixOperations.multiply_with_scalar(krod,constant1);
kbeam=MatrixOperations.multiply_with_scalar(kbeam,constant2);
      double [][]kebc=MatrixOperations.add(krod,kbeam);
      double [][]constant3={{x21/L,y21/L,0,0,0,0},{-
y21/L,x21/L,0,0,0,0},{0,0,1,0,0,0},
                  {0,0,0,x21/L,y21/L,0},{0,0,0,-
y21/L,x21/L,0},{0,0,0,0,0,1}};
      double
[][]ke=MatrixOperations.multiply(MatrixOperations.transpose(te),(
MatrixOperations.multiply(kebc,constant3)));
      return ke;
   }
```

Figure 5.4. Beam3Stiff module

5.3. Implementation of Global Structural Stiffness Matrix

The Java code shown in Figure 5.5 called PBGlobStrStiff returns global structural matrix of beam-column frame structures. Global structural stiffness matrix,"K^G" is a square matrix which accommodates all displacement force relations of every degree of freedoms. Since plane beam elements have 3 freedoms in every node, the total number of the freedom is equal to;

$$\Sigma n_{freedom} = \text{total number of freedom} = 2 * \text{number of nodes} \quad (5.7)$$

Thus, global structural stiffness matrix of beam-column frame structures has ($\Sigma n_{freedom} * \Sigma n_{freedom}$) dimensions.

```
public static double [][] PBGlobStrStiff(double [][]allXYcoord,int
[][]elempositions,double []ElasticMs, double areas, double InertiaZ)
   {int numberofele=elempositions.length;
      int numnod=allXYcoord.length;
      double [][]K=MatrixOperations.zero_matrix(3*numnod);
      for(int e=1;e<=numberofele;e++)
      {int []eNL=elempositions[e-1];
         int ni=eNL[0];
         int nj=eNL[1];
         int []ongfitab=new int[6];
         ongfitab[0]=3*ni-2;       ongfitab[1]=3*ni-1;
         ongfitab[2]=3*ni;          ongfitab[3]=3*nj-2;
         ongfitab[4]=3*nj-1;       ongfitab[5]=3*nj;
         double [][]Xycoord=new double[2][];
         Xycoord[0]=allXYcoord[ni-1];
         Xycoord[1]=allXYcoord[nj-1];
         double  [][]Ke=Beam3Stiff(Xycoord,ElasticMs[e-1],areas[e-
1],InertiaZ[e-1]);
         for(int i=1;i<=Ke.length;i++)
         {
            int ii=ongfitab[i-1];
            for(int j=i;j<=Ke.length;j++)
            {int jj=ongfitab[j-1];
               K[ii-1][jj-1]+=Ke[i-1][j-1];
               K[jj-1][ii-1]=K[ii-1][jj-1];}
         }
      }
      return K;}
```

Figure 5.5. PBGlobStrStiff module

5.3.1. Formal Parameters of PBGlobStrStiff

allXYcoord: All global coordinates of the system joints
{(x1, y1), (x2, y2)…….(x_{last}, y_{last})}

elempositions: Element end points that are assigned for the system joints which are required to define an element and its local axis.
{(#1, #2)$_1$, (#1, #2)$_2$….. (#1, #2)$_{last}$}

ElasticMs: Elastic Modulus of each element in the same order of "elemposition" parameter (E_1,E_2….E_{last})

areas: Cross-sections of each element in the same order of "elemposition" parameter (A_1,A_2….A_{last})

InertiaZ: Moment of inertia about the neutral axis (I_{zz})

PBGlobStrStiff module returns $\Sigma n_{freedom} * \Sigma n_{freedom}$ global structural stiffness matrix of the beam frames.

5.4. Implementation of Reduced Global Structural Stiffness System

Displacement boundary conditions of the plane beam systems have to be applied into the linear system of equations to prevent singularity. The computer implementation of the boundary conditions is carried out by the ReducedGlobStrStiff module which is given in Figure 3.7. The formal parameters of the ReducedGlobStrStiff module are repeated for convenience.

Nboolean: Freedom information which contains boundary conditions

K: Global structural stiffness matrix, "K^G", which reduction is needed.

ReducedGlobStrStiff module returns $\Sigma n_{freedom} * \Sigma n_{freedom}$ reduced global structural stiffness matrix of the beam frame structures.

5.5. Implementation of External Loads

External loads which are defined by Nodevalues parameter are rearranged with sub module called ReducedNodeForces which is shown in Figure 5.6. The purpose of this arrangement is to eliminate the given node forces at joints considering the joint constrains defined in Nboolean parameter.

5.5.1. Formal Parameters of PBReducedNodeForces

Nboolean: Freedom information which contains boundary conditions

Nodevalues: Initial values of the forces in the same order of Nboolean.

```
public static double [] PBReducedNodeForces
(boolean [][]Nboolean, double [][]NodeValues)
   {
      double []freduced=new

double[NodeValues.length*NodeValues[0].length];
      for(int i=0,k=0;i<NodeValues.length;i++)
      {
         for(int j=0;j<NodeValues[i].length;j++)
         {
            freduced[k]=NodeValues[i][j];
            k++;
         }
      }
      for(int n=1;n<=Nboolean.length;n++)
      {
         for(int j=1;j<=3;j++)
         {
            if(Nboolean[n-1][j-1])
            {
               freduced[3*(n-1)+j-1]=0;
            }
         }
      }
      return freduced;
   }
```

Figure 5.6. PBReducedNodeForces module

5.6. Implementation of Displacement Solutions

PbSolution module shown in Figure 5.7 solves the linear system of equations of the beam-column frame systems by using equation (3.11).

```
public static Object [] PBSolution(double [][]allXYcoord,int
[][]elempositions,double []ElasticMs, double areas,double InetriaZ
          boolean [][]Nboolean, double [][]Nodevalues)
   {double [][]K=KeCalculation.PBGlobStrStiff(allXYcoord,
elempositions,ElasticMs,areas,InertiaZ);
       double
[][]Kreduced=KeCalculation.PBReducedGlobStrStiff(Nboolean,K);
       double []f=new
double[Nodevalues.length*Nodevalues[0].length];
       for(int i=0,k=0;i<Nodevalues.length;i++)
       {for(int j=0;j<Nodevalues[i].length;j++)
          { f[k]=Nodevalues[i][j];  k++;}}
double []freduced=KeCalculation.PBReducedNodeForces
(Nboolean,Nodevalues);
       Jama.Matrix u=(new Jama.Matrix(Kreduced)).solve(new
Jama.Matrix(freduced,Kreduced.length));
       double []my_u=new double[Kreduced.length];
       for(int i=0;i<my_u.length;i++)
          my_u[i]=(u.getArray())[i][0];
       for(int i=0;i<K.length;i++)
       {
          f[i]=MatrixOperations.dot_product(K[i],my_u);
       }
```

Figure 5.7. PBSolution module

5.6.1. Formal Parameters of PBSolution

allXYcoord: All global coordinates of the system joints
{(x1, y1), (x2, y2)…….(x_{last}, y_{last})}

elempositions: Element end points that are assigned for the system joints. It is required to define an element and its local axis.
{(#1, #2)$_1$, (#1, #2)$_2$….. (#1, #2)$_{last}$}

ElasticMs: Elastic Modulus of each element in the same order of "elemposition" parameter (E_1,E_2….E_{last})

areas: Cross-sections of each element in the same order of "elemposition" parameter $(A_1, A_2 \ldots A_{last})$

InertiaZ: Moment of inertia about the neutral axis (I_{zz})

Nboolean: Freedom information which contains boundary conditions

Nodevalues: Initial values of the forces or prescribed displacements in the same order of Nboolean.

PBSolution module returns the unknown displacements of each node. Local parameter "my_u" is assigned for the displacement solutions. The solutions stored in "my_u" matrix are in the order of , u_{x1}, $u_{y1,}\theta_1$.

5.7. Implementation of Axial Forces and Bending Moments

Module PBIntForces given in Figure 5.8, computes the internal forces of the beam elements.

5.7.1. Formal Parameters of BPIntForces

allXYcoord: All global coordinates of the system joints
$\{(x1, y1), (x2, y2) \ldots\ldots (x_{last}, y_{last})\}$

elempositions: Element end points that are assigned for the system joints. It is required to define an element and its local axis.
$\{(\#1, \#2)_1, (\#1, \#2)_2 \ldots\ldots (\#1, \#2)_{last}\}$

ElasticMs: Elastic Modulus of each element in the same order of "elemposition" parameter $(E_1, E_2 \ldots E_{last})$

areas: Cross-sections of each element in the same order of "elemposition" parameter $(A_1, A_2 \ldots A_{last})$

InertiaZ: Moment of inertia about the neutral axis (I_{zz})

u: Displacements of the nodes solved by Jama Matrix module $(u_{x1}, u_{y1}, \theta_1 \ldots u_{xlast,}, u_{ylast,}\theta_{last})$

BPIntForces module returns not only the axial forces of the beam elements but also the bending moment values of the end nodes of the beam-column elements.

```
public static double [][] PBIntForces(double [][]allXYcoord,
int [][]elempositions,double []ElasticMs,double areas,
double InertiaZ,double []u)
   {
      int numnod=allXYcoord.length;
      int numberofele=elempositions.length;
      double [][]p=MatrixOperations.zero_matrix(numberofele,3);
      double []elemdisp=MatrixOperations.zero_vector(6);
      for(int e=1;e<=numberofele;e++)
      {
         int []eNL=elempositions[e-1];
         int ni=eNL[0];
         int nj=eNL[1];
         double [][]Xycoord=new double[2][];
         Xycoord[0]=allXYcoord[ni-1];
         Xycoord[1]=allXYcoord[nj-1];
         int []ongfitab=new int[6];
         ongfitab[0]=3*ni-2; ongfitab[1]=3*ni-1; ongfitab[2]=3*ni;
ongfitab[3]=3*nj-2;
         ongfitab[4]=3*nj-1;ongfitab[5]=3*nj;
         for(int i=1;i<=6;i++)
         { int ii=ongfitab[i-1]; elemdisp[i-1]=u[ii-1];
         }
         p[e-1]=Beam3IntForces(Xycoord,ElasticMs[e-1],
areas[e-1],InertiaZ[e-1],elemdisp);
      }
      return p;
   }
   public static double[] Beam3IntForces(double [][]Xycoord,
double ElasticM,double area,double InertiaZ,double []elemdisp)
   {
      double []pe=MatrixOperations.zero_vector(3);
      double x1=Xycoord[0][0];double y1=Xycoord[0][1];
      double x2=Xycoord[1][0];double y2=Xycoord[1][1];
      double x21=x2-x1;double y21=y2-y1;
      double LL=x21*x21+y21*y21;
      double L=Math.sqrt(LL);
      pe[0]=ElasticM*area*(x21*(elemdisp[3]-
elemdisp[0])+y21*(elemdisp[4]-elemdisp[1]))/LL;
      double duy=(x21*(elemdisp[4]-elemdisp[1])-
y21*(elemdisp[3]-elemdisp[0]))/L;
      double cnst1=6*duy/LL+2*(-2*elemdisp[2]-elemdisp[5])/L;
      double cnst2=-6*duy/LL+2*(elemdisp[2]+2*elemdisp[5])/L;
      pe[1]=ElasticM*InertiaZ*cnst1;
      pe[2]=ElasticM*InertiaZ*cnst2;
      return pe;
   }}
```

Figure 5.8. PBIntForces module

5.8. Implementation of Results

Java object called result2 which is shown in Figure 5.9 collects the solutions of three dimensional truss problems. Results of the system are gathered with the three local parameters and transmitted to the Output Handler.

Local parameter “dissolved” collects the displacement solution matrix which is evaluated by Jama.Matrix module. Local parameter “NForces” collects reaction forces of the system by applying equation (3.14). Finally, local parameter “Eintforces” collects internal forces of the elements by using equation (2.80).

```
Jama.Matrix u=(new Jama.Matrix(Kreduced)).
solve(new Jama.Matrix(fmod,Kreduced.length));
      double []my_u=new double[Kreduced.length];
      for(int i=0;i<my_u.length;i++)
         my_u[i]=(u.getArray())[i][0];
      for(int i=0;i<K.length;i++)
      {
         f[i]=MatrixOperations.dot_product(K[i],my_u);
      }
      double [][]elefor=NodeForces.PBIntForces(allXYcoord,
elempositions,ElasticMs,areas,InertiaZ,
my_u);

Object []result2=new Object[3];
      result[0]=dissolved;
      result[1]=Nforces;
      result[2]=Eintforces;
      return result;
```

Figure 5.9. Result2 object for beam-column frames

5.9. Flow Chart of the Beam-Column Frame Program

Schematic representation of the algorithm of the beam-column truss program is given in Figure 5.10.

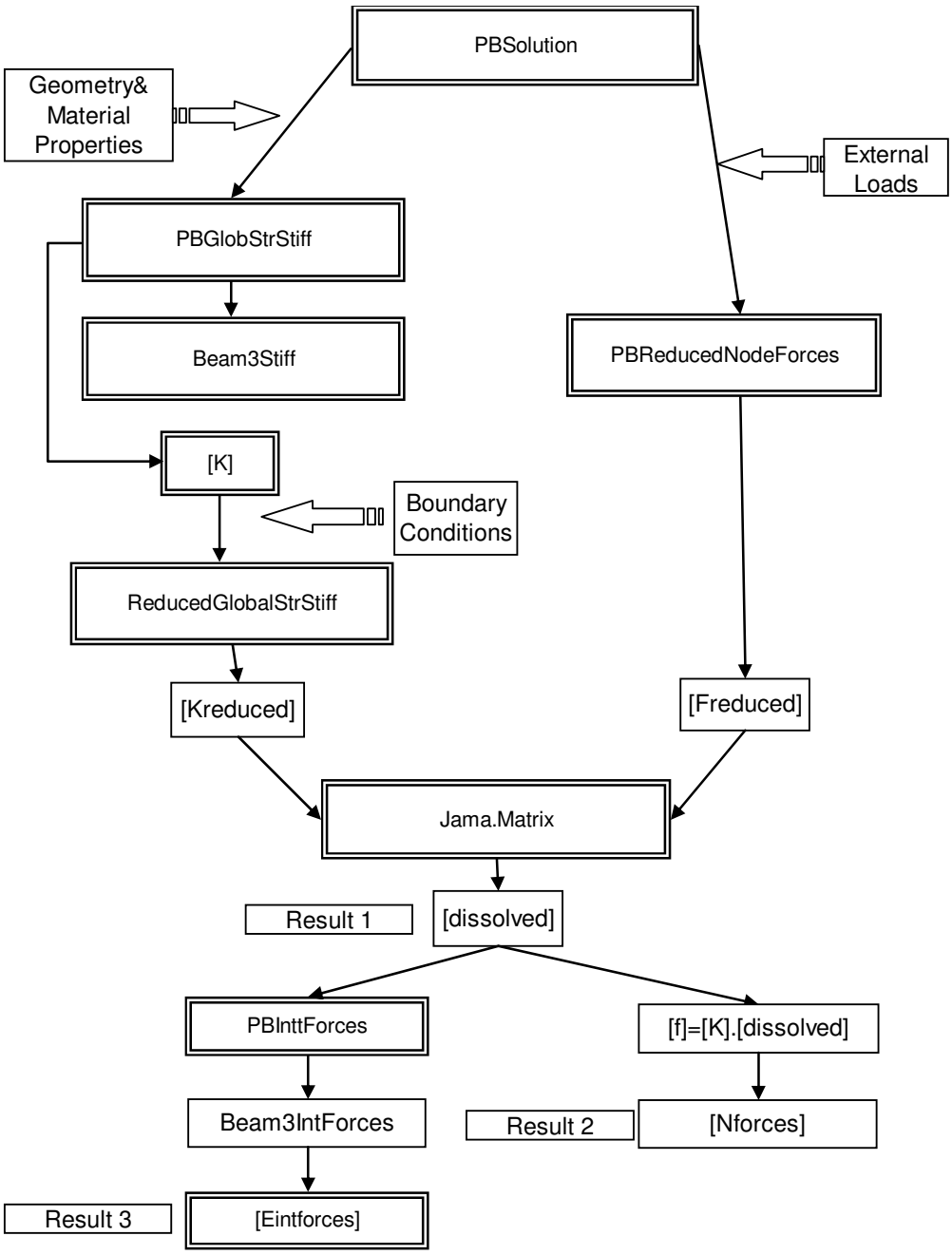


Figure 5.10. Flowchart of the beam-column frame program

5.10. Case Study of a Beam-Column Frame Problem

The purpose of this section is to compare the Java beam-colum program with the SAP2000. A simple beam frame problem which is shown in Figure 5.11 is used to test the algorithm. The frame structure has 21 members and 22 nodes with 21 joint loads applying at the sketched points. The properties of the problem which are prepared by Microsoft Excel program are given in the Table 5.1.

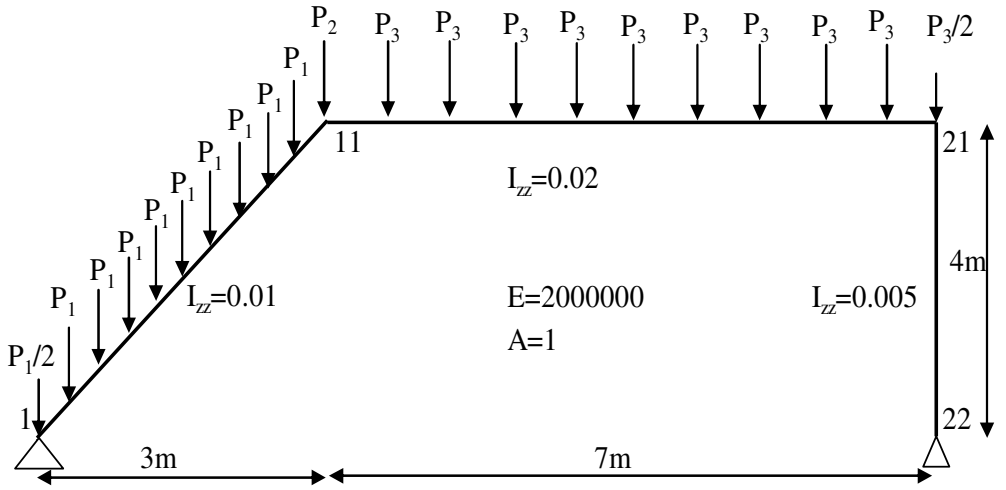


Figure 5.11. Beam-column problem

Member internal forces and the Node Deflections of the given plane-beam problem are calculated by Java beam-column program and Sap2000. The deflections and the internal force solutions are compared in Table 5.2 and Table 5.3.

It is shown that Java beam-column program evaluates the same results with Sap2000 for the frame introduced.

Table 5.1. Properties of the beam frame problem

Ref No	Coordinates		Element End Nodes		Elastic Modulus	Areas	I_{zz}	Node Constraints			Joint Loads		
	x	y	i	j				x	y	θ	x	y	m
1	0	0	1	2	2000000	1	0.01	1	1	0	0	-0.3	0
2	0.3	0.4	2	3	2000000	1	0.01	0	0	0	0	-0.6	0
3	0.6	0.8	3	4	2000000	1	0.01	0	0	0	0	-0.6	0
4	0.9	1.2	4	5	2000000	1	0.01	0	0	0	0	-0.6	0
5	1.2	1.6	5	6	2000000	1	0.01	0	0	0	0	-0.6	0
6	1.5	2	6	7	2000000	1	0.01	0	0	0	0	-0.6	0
7	1.8	2.4	7	8	2000000	1	0.01	0	0	0	0	-0.6	0
8	2.1	2.8	8	9	2000000	1	0.01	0	0	0	0	-0.6	0
9	2.4	3.2	9	10	2000000	1	0.01	0	0	0	0	-0.6	0
10	2.7	3.6	10	11	2000000	1	0.01	0	0	0	0	-0.6	0
11	3	4	11	12	2000000	1	0.02	0	0	0	0	-1	0
12	3.7	4	12	13	2000000	1	0.02	0	0	0	0	-1.4	0
13	4.4	4	13	14	2000000	1	0.02	0	0	0	0	-1.4	0
14	5.1	4	14	15	2000000	1	0.02	0	0	0	0	-1.4	0
15	5.8	4	15	16	2000000	1	0.02	0	0	0	0	-1.4	0
16	6.5	4	16	17	2000000	1	0.02	0	0	0	0	-1.4	0
17	7.2	4	17	18	2000000	1	0.02	0	0	0	0	-1.4	0
18	7.9	4	18	19	2000000	1	0.02	0	0	0	0	-1.4	0
19	8.6	4	19	20	2000000	1	0.02	0	0	0	0	-1.4	0
20	9.3	4	20	21	2000000	1	0.02	0	0	0	0	-1.4	0
21	10	4	21	22	2000000	1	0.005	0	0	0	0	-0.7	0
22	10	0						1	1	0	0	0	0

Table 5.2. Comparison of internal forces of beam elements

	SAP2000			JAVA					
Member	**P** (kN)	**M_i** (kN.m)	**M_j** (kN.m)	**P** (kN)	**M_i** (kN.m)	**M_j** (kN.m)	**Comparison of Results(%)**		
1	-9.742	0.000	1.589	-9.745	0.000	1.587	100.0	100.0	100.0
2	-9.262	1.589	2.998	-9.265	1.587	2.994	100.0	99.9	99.9
3	-8.782	2.998	4.227	-8.785	2.994	4.221	100.0	99.9	99.8
4	-8.302	4.227	5.276	-8.305	4.221	5.267	100.0	99.8	99.8
5	-7.822	5.276	6.145	-7.825	5.267	6.134	100.0	99.8	99.8
6	-7.342	6.145	6.834	-7.345	6.134	6.821	100.0	99.8	99.8
7	-6.862	6.834	7.343	-6.865	6.821	7.328	100.0	99.8	99.8
8	-6.382	7.343	7.672	-6.385	7.328	7.655	99.9	99.8	99.8
9	-5.902	7.672	7.821	-5.905	7.655	7.802	99.9	99.8	99.8
10	-5.422	7.821	7.790	-5.425	7.802	7.769	99.9	99.8	99.7
11	-3.303	7.790	10.100	-3.308	7.769	10.079	99.8	99.7	99.8
12	-3.303	10.765	11.430	-3.308	10.779	11.409	99.8	99.9	99.8
13	-3.303	11.430	11.780	-3.308	11.409	11.759	99.8	99.8	99.8
14	-3.303	11.780	11.150	-3.308	11.759	11.129	99.8	99.8	99.8
15	-3.303	11.150	9.540	-3.308	11.129	9.519	99.8	99.8	99.8
16	-3.303	9.540	6.950	-3.308	9.519	6.929	99.8	99.8	99.7
17	-3.303	6.950	3.380	-3.308	6.929	3.359	99.8	99.7	99.4
18	-3.303	3.380	-1.170	-3.308	3.359	-1.191	99.8	99.4	98.2
19	-3.303	-1.170	-6.700	-3.308	-1.191	-6.721	99.8	98.2	99.7
20	-3.303	-6.700	-13.210	-3.308	-6.721	-13.231	99.8	99.7	99.8
21	-10.000	-13.210	0.000	-10.000	-13.231	0.000	100.0	100.0	100.0

Table 5.3. Comparison of node deflections

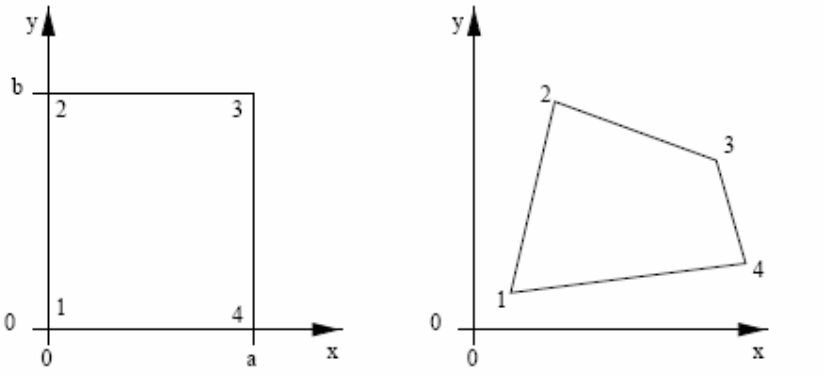
	SAP2000			JAVA					
Joint	**Ux** (10^{-4}m)	**Uy** (10^{-4}m)	**R3** Radians	**U1** (10^{-4}m)	**U2** (10^{-4}m)	**R3** Radians	**Comparison of Results (%)**		
1	0	0.000	-0.00164	0.000	0.000	-0.00164	100.0	100.0	100.0
2	0.650	-0.496	-0.00162	0.650	-0.491	-0.00162	100.0	98.9	99.6
3	1.284	-0.980	-0.00157	1.284	-0.969	-0.00156	100.0	98.9	99.6
4	1.89	-1.442	-0.00147	1.889	-1.426	-0.00147	100.0	98.9	99.6
5	2.453	-1.872	-0.00136	2.452	-1.851	-0.00135	100.0	98.9	99.5
6	2.964	-2.263	-0.00121	2.963	-2.236	-0.00121	100.0	98.8	99.5
7	3.414	-2.607	-0.00105	3.413	-2.576	-0.00104	100.0	98.8	99.4
8	3.796	-2.900	-0.00087	3.795	-2.864	-0.00087	100.0	98.8	99.4
9	4.106	-3.137	-0.00069	4.104	-3.098	-0.00068	100.0	98.8	99.2
10	4.339	-3.318	-0.00049	4.337	-3.275	-0.00049	100.0	98.7	99.1
11	4.494	-3.439	-0.0003	4.492	-3.393	-0.00029	100.0	98.7	98.6
12	4.492	-3.594	-0.00014	4.491	-3.546	-0.00014	100.0	98.7	97.6
13	4.489	-3.628	0.000048	4.490	-3.577	4.83E-05	100.0	98.6	99.4
14	4.487	-3.524	0.000251	4.489	-3.470	0.000254	100.0	98.5	98.8
15	4.485	-3.277	0.000452	4.488	-3.222	0.000454	99.9	98.3	99.5
16	4.482	-2.896	0.000633	4.486	-2.839	0.000635	99.9	98.0	99.7
17	4.48	-2.337	0.000777	4.485	-2.341	0.000779	99.9	99.8	99.8
18	4.478	-1.759	0.000867	4.484	-1.761	0.000869	99.9	99.9	99.8
19	4.475	-1.140	0.000887	4.483	-1.141	0.000888	99.8	99.9	99.9
20	4.473	-0.539	0.000818	4.482	-0.539	0.000819	99.8	100.0	99.9
21	4.471	-0.020	0.000644	4.481	-0.020	0.000644	99.8	100.0	100.0
22	0	0.000	-0.002	0.000	0.000	-0.002	100.0	100.0	100.0

6. JAVA IMPLEMANTATION OF QUADRILATERAL ELEMENTS

6.1. Introduction

In the case of two dimensional finite element discretisation differential equations are to be solved over regions. This is accomplished by defining general triangular and quadrilateral elements with the concept of local coordinate systems.

In geometry, a quadrilateral is a polygon with four sides or edges and four vertices or corners. The best-known quadrilaterals are the square, with four internal right angles, four sides of equal length, and four axes of symmetry; the rectangle, with four internal right angles, opposite sides that are equal in length, and two axes of symmetry; the rhombus, with four equal sides and two axes of symmetry; the parallelogram, with two pairs of parallel sides that are equal in length; and the trapezoid, with two parallel sides of unequal length. Figure 6.1 shows a rectangular element and a more general quadrilateral element.



(a) Plane rectangular element (b) General quadrilateral element

Figure 6.1. Types of quadrilateral element

In order to implement stiffness matrix of a general quadrilateral element which is given by the equation (2.94), shape function matrix $[N]$ is required. Shape function is an approximation polynomial which equals to the function it approximates at a number of specified stations or points. A shape function is written for each individual node of a two dimensional finite element and has the property that its magnitude is one at that node and

zero for all other nodes in that element. The shape functions of simple 4 node rectangular element shown in Figure 6.1(a) were first derived by Taig and given in equation (6.1) [14].

$$\begin{aligned} N_1 &= (1-x/a)(1-y/a) \\ N_2 &= (1-x/a)(y/b) \\ N_3 &= (x/a)(y/b) \\ N_4 &= (x/a)(1-y/b) \end{aligned} \tag{6.1}$$

Constructing similar shape functions in the global coordinates (x,y) for the general quadrilateral results in very complex algebraic expressions. Instead it is better to work in a local coordinate system as shown in Figure 6.2.

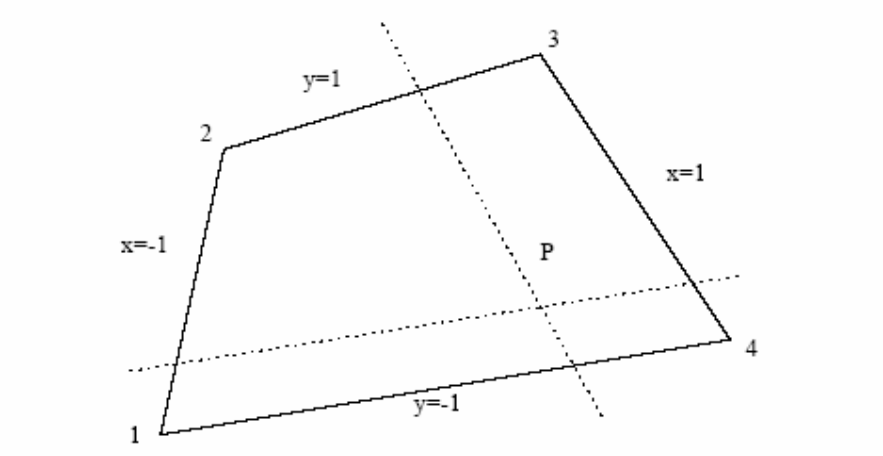


Figure 6.2. Local coordinate system for quadrilateral

The normalized coordinates x and y vary from -1 on one side to +1 at the other side ,taking the value zero over the quadrilateral midpoints. In this system the intersection of the bisectors of opposite sides of the quadrilateral is the point (0,0), while the corners 1,2,3,4 are (-1,-1),(-1,1),(1,1),(1,-1), respectively. Instead of using x and y for local coordinates of quadrilateral elements, natural coordinates of the element are denoted by ξ and η. When this natural coordinate system is adopted, the shape functions for any quadrilateral elements take the simple form and shape function matrix $[N]$ can easily be evaluated. However, it was shown in equation (2.93) that two dimensional element properties involve not only $[N]$ but also their derivatives with respect to the global coordinates x and y. In order to evaluate strain displacement matrix $[B]$, given by equation

(2.93), partial derivatives of shape functions with respect to Cartesian coordinates x and y are required. Thus, the function which defines the transformation from the local natural coordinates (ξ, η) to global coordinates (x,y) for any quadrilateral can be evaluated by using equation (6.2).

$$x = \sum_{k=1}^{m} F_k(\xi_k, \eta_k) x_k \, , \, y = \sum_{k=1}^{m} F_k(\xi_k, \eta_k) y_k \, \text{z} \tag{6.2}$$

Here m is the number of points defining the geometry of the element and $F_k(\xi_k, \eta_k)$ is the transformation function. Since the construction and derivation of these transformation functions that satisfy consistency requirements for higher order elements becomes complicated, the concept of isoparametric elements is introduced in the late 1960s by Ergatoudis and Zienkiewics [15].

The shape functions which introduced in equation (2.92) were used to describe the variation of the unknown displacements in an element. The generalization of displacement interpolation of an arbitrary two dimensional element with n nodes in terms of the shape functions can be written as;

$$u_x = \sum_{i=1}^{n} u_{xi} N_i^e \, , u_y = \sum_{i=1}^{n} u_{yi} N_i^e \tag{6.3}$$

The main idea to define an isoparametric element is to use the same shape functions to specify the relation between the global (x,y) and local (ξ,η) coordinate systems. Therefore, the coordinate transformation of any element is equals to:

$$x = \sum_{i=1}^{n} x_i N_i^e \, , \, y = \sum_{i=1}^{n} y_i N_i^e \tag{6.4}$$

In order to construct $[B]$, the strain-displacement stiffness matrix of quadrilateral elements evaluated in equation (2.93), the partial derivatives of the shape functions are needed.

6.2. Computation of Strain-Displacement Matrix

Since the shape functions of an isoparametric quadrilateral element are not directly functions of Cartesian coordinates, differentials of natural and Cartesian coordinates can be evaluated by Jacobian transformations.

6.2.1. The Jacobian Transformation

The Jacobian matrix is the matrix of all first-order partial derivatives of a vector-valued function. The Jacobian matrix of shape function is evaluated by applying chain rule as

$$\begin{bmatrix} dx \\ dy \end{bmatrix} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial x}{\partial \eta} \\ \frac{\partial y}{\partial \xi} & \frac{\partial y}{\partial \eta} \end{bmatrix} \begin{bmatrix} d\xi \\ d\eta \end{bmatrix} = J^T \begin{bmatrix} d\xi \\ d\eta \end{bmatrix}, \quad \begin{bmatrix} d\xi \\ d\eta \end{bmatrix} = \begin{bmatrix} \frac{\partial \xi}{\partial x} & \frac{\partial \xi}{\partial y} \\ \frac{\partial \eta}{\partial x} & \frac{\partial \eta}{\partial y} \end{bmatrix} \begin{bmatrix} dx \\ dy \end{bmatrix} = J^{-T} \begin{bmatrix} dx \\ dy \end{bmatrix} \tag{6.5}$$

In quadrilateral element derivations, bi-dimensional Jacobian transformation is used to associate the differentials of (x, y) to the differentials of (ξ,η). The Jacobian Matrix of (x, y) with respect to (ξ,η) is denoted as $[J]$, and the Jacobian Matrix of (ξ,η) with respect to (x, y) is denoted as $\left[J^{-1}\right]$.

$$J = \frac{\partial(x,y)}{\partial(\xi,\eta)} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix} = \begin{bmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{bmatrix}, \; J^{-1} = \frac{\partial(\xi,\eta)}{\partial(x,y)} = \begin{bmatrix} \frac{\partial \xi}{\partial x} & \frac{\partial \eta}{\partial x} \\ \frac{\partial \xi}{\partial y} & \frac{\partial \eta}{\partial y} \end{bmatrix} = \frac{1}{A} \begin{bmatrix} J_{22} & -J_{12} \\ -J_{21} & J_{11} \end{bmatrix} \tag{6.6}$$

For simplicity, $[J]$ and $\left[J^{-1}\right]$ are called Jacobian and inverse Jacobian, respectively [12]. The determinant of Jacobian gives the scalar A and equals to;

$$A = |J| = \det(J) = J_{11}J_{22} - J_{12}J_{21} \tag{6.7}$$

6.2.2. Shape Function Derivatives

The shape functions of quadrilateral elements in terms of the quadrilateral coordinates defined by natural coordinates ξ and η. The derivatives with respect to x and y are given in the following equations.

$$\frac{\partial N_i^e}{\partial x} = \frac{\partial N_i^e}{\partial \xi}\frac{\partial \xi}{\partial x} + \frac{\partial N_i^e}{\partial \eta}\frac{\partial \eta}{\partial x}, \qquad \frac{\partial N_i^e}{\partial y} = \frac{\partial N_i^e}{\partial \xi}\frac{\partial \xi}{\partial y} + \frac{\partial N_i^e}{\partial \eta}\frac{\partial \eta}{\partial y} \tag{6.8}$$

Equations given in (6.8) also can be written in a matrix form;

$$\begin{bmatrix} \frac{\partial N_i^e}{\partial x} \\ \frac{\partial N_i^e}{\partial y} \end{bmatrix} = \begin{bmatrix} \frac{\partial \xi}{\partial x} & \frac{\partial \eta}{\partial x} \\ \frac{\partial \xi}{\partial y} & \frac{\partial \eta}{\partial y} \end{bmatrix} \begin{bmatrix} \frac{\partial N_i^e}{\partial \xi} \\ \frac{\partial N_i^e}{\partial \eta} \end{bmatrix} = \frac{\partial(\xi,\eta)}{\partial(x,\mathrm{y})} \begin{bmatrix} \frac{\partial N_i^e}{\partial \xi} \\ \frac{\partial N_i^e}{\partial \eta} \end{bmatrix} = J^{-1} \begin{bmatrix} \frac{\partial N_i^e}{\partial \xi} \\ \frac{\partial N_i^e}{\partial \eta} \end{bmatrix} \tag{6.9}$$

6.2.3. Calculating Jacobian Matrix

Geometric relation between shape functions and corner coordinates are defined in equation (6.4) which is repeated for convenience:

$$x = \sum_{i=1}^{n} x_i N_i^e \qquad \mathrm{y} = \sum_{i=1}^{n} y_i N_i^e \tag{6.10}$$

In order to calculate the Jacobian Matrix values at any quadrilateral point, first shape functions are differentiated with respect to the quadrilateral coordinates.

$$\frac{\partial x}{\partial \xi} = \sum_{i=1}^{n} x_i \frac{\partial N_i^e}{\partial \xi}, \qquad \frac{\partial y}{\partial \xi} = \sum_{i=1}^{n} y_i \frac{\partial N_i^e}{\partial \xi}, \qquad \frac{\partial x}{\partial \eta} = \sum_{i=1}^{n} x_i \frac{\partial N_i^e}{\partial \eta}, \qquad \frac{\partial y}{\partial \eta} = \sum_{i=1}^{n} y_i \frac{\partial N_i^e}{\partial \eta} \tag{6.11}$$

Since x_i and y_i do not depend on ξ and η; Jacobian matrix can be written as

$$J=\begin{bmatrix} J_{11} & J_{12} \\ J_{21} & J_{22} \end{bmatrix}=\begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix}=\begin{bmatrix} \frac{\partial N_1^e}{\partial \xi} & \frac{\partial N_2^e}{\partial \xi} & \cdots & \frac{\partial N_n^e}{\partial \xi} \\ \frac{\partial N_1^e}{\partial \eta} & \frac{\partial N_2^e}{\partial \eta} & \cdots & \frac{\partial N_n^e}{\partial \eta} \end{bmatrix}\begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ . & . \\ . & . \\ . & . \\ x_n & y_n \end{bmatrix} \tag{6.12}$$

By using equation 6.12, Jacobian matrix values can be evaluated at any given ξ and η points. The strain-displacement matrix [B] can be constructed by computing [J] and applying chain rule introduced in equation (6.9). Finally, the stiffness matrix of two dimensional elements can be evaluated by integrating equation (2.94) over the element domain. Required integration will be done by Gauss numerical integration method.

Computer implementation of isoparametric quadrilateral elements for the plane stress problems covers the computation of the element stiffness matrix and calculation of stresses at selected points.

6.2.4. Derivation of Stiffness Matrix

The stiffness matrix of a two dimensional element for unit thickness is given by the equation (2.94), which is repeated here;

$$[k_m]=\iint [B]^T [D][B]\,dxdy \tag{6.13}$$

Here [B] is the strain-displacement matrix, [D] is the stress-strain matrix. Considering the quadrilateral element natural coordinates, the stiffness matrix equation can be transformed to the canonical form, that is;

$$[k_m]=\int_{-1}^{1}\int_{-1}^{1} F(\xi,\eta)d\xi d\eta \tag{6.14}$$

Since the differentials of Cartesian coordinates and natural coordinates dependent with the following equation;

$$dxdy = \det[J]\, d\xi d\eta \tag{6.15}$$

$F(\xi,\eta)$, function defined in canonical form can be expressed as

$$F(\xi,\eta) = [B]^T [D][B]\det[J] \tag{6.16}$$

6.3. Implementation of Gauss Rule for Quadrilateral Elements

In numerical analysis, numerical integration constitutes a broad family of algorithms for calculating the numerical value of a definite integral. The basic problem considered by numerical integration is to compute an approximate solution to a definite integral:

$$\int_a^b f(x)dx \tag{6.17}$$

One of the most effective numerical integration methods is Gaussian quadrature which achieves approximately accurate solutions by using minimal number of sample points. Therefore, for evaluations of integrals derived in equation (6.14), Gauss quadrature method will be used. Gauss quadrature method is an approximation of the definite integral of a function, usually stated as a weighted sum of function values at specified points within the domain of integration. An *n*-point Gaussian quadrature rule, named after Carl Friedrich Gauss, is a quadrature rule constructed to yield an exact result for polynomials of degree $2n - 1$, by a suitable choice of the n points x_i and weights w_i. The domain of integration for such a rule is conventionally taken as $[-1, 1]$, so the rule is stated as

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=1}^{n} \omega_i f(x_i) \tag{6.18}$$

The evaluation points are the roots of a polynomial belonging to a class of orthogonal polynomials named as "Legendre polynomials", $P_n(x)$. With the n^{th} polynomial normalized to give $P_n(1) = 1$, the i^{th} Gauss node, x_i, is the i^{th} root of P_n; its weight is given by Abramowitz & Stegun [16].

$$w_i = \frac{2}{(1-x_i^2)(P_n'(x_i))^2} \tag{6.19}$$

The Java code shown in Figure 6.3 called Gauss1D returns x_i and w_i values for the first five unidimensional Gauss rules.

6.3.1. Formal Parameters of Gauss1D:

NumPnt: The number of points to calculate the given integral
{An integer number between 1 and 5}

Order: The point on which information is required
{An integer number between 1 and 5}

One dimensional Gauss rule entries for solving the definite integration problem are listed in Table 6.1.

Table 6.1. Abscissa and weights for Gaussian quadrature rules

Number of Points	Points (x_i)	Weights (w_i)
1	0	2
2	$\pm\sqrt{1/3}$	1
3	0	8/9
	$\pm\sqrt{3/5}$	5/9
4	$\pm\sqrt{(3-2\sqrt{6/5})/7}$	$\frac{18+\sqrt{30}}{36}$
	$\pm\sqrt{(3+2\sqrt{6/5})/7}$	$\frac{18-\sqrt{30}}{36}$
5	0	128/225
	$\pm\frac{1}{3}\sqrt{5-2\sqrt{10/7}}$	$\frac{322+13\sqrt{70}}{900}$
	$\pm\frac{1}{3}\sqrt{5+2\sqrt{10/7}}$	$\frac{322-13\sqrt{70}}{900}$

The values of required point abscissa and integration weight are directly implemented to Gauss1D module which is shown in Figure 6.3.

```
Public static double[] Gauss1D(int NumPnt, int Order)
   {
      double []point2={-1/Math.sqrt(3),1/Math.sqrt(3)};
      double []weight3={5.0/9,8.0/9,5.0/9};
      double []point3={-Math.sqrt(3.0/5),0,Math.sqrt(3.0/5)};
      double []weight4={(1.0/2)-(Math.sqrt(5.0/6)/6), (1.0/2)+
        (Math.sqrt(5.0/6)/6),(1.0/2)+(Math.sqrt(5.0/6)/6), (1.0/2)-
        (Math.sqrt(5.0/6)/6)};double []g4={-
         Math.sqrt((3+2*Math.sqrt(6.0/5))/7),-Math.sqrt((3-2*
         Math.sqrt(6.0/5))/7), Math.sqrt((3-2*Math.sqrt(6.0/5))/7),
         Math.sqrt((3+2*Math.sqrt(6.0/5))/7)};
      double []point5={-Math.sqrt(5+2*Math.sqrt(10.0/7))/3,-
        Math.sqrt(5-2*Math.sqrt(10.0/7))/3,0, Math.sqrt(5-2*
       Math.sqrt(10.0/7))/3, Math.sqrt(5+2*Math.sqrt(10.0/7))/3};
      double []weight5={(322-13*Math.sqrt(70))/900,(322+13*
         Math.sqrt(70))/900,512.0/900,(322+13*
         Math.sqrt(70))/900,(322-13*Math.sqrt(70))/900};
      Order--;
      if(NumPnt==1)
  {double []result= {0,2};return result; }
      else if(NumPnt==2)
  {double []result= {point2[Order],1};return result; }
      else if(NumPnt==3)
  {double []result= {point3[Order],weight3[Order]}; return result; }
      else if(NumPnt==4)
  {double []result= {point4[Order],weight4[Order]}; return result; }
      else if(NumPnt==5)
      {double []result= {point5[Order],weight5[Order]}; return
result;}
      else
         return null;
    }
```

Figure 6.3. Gauss1D module.

6.3.2. Example Commands for Gauss1D:

The command of Gauss1D [3, 1], calls three point gauss rule and x_i and w_i values of first point and returns x_1=-0.7746 and w_1=0.5556. The command of Gauss1D [3, 2], calls three point gauss rule and x_i and w_i values of second point and returns x_2=0 and

w_2=0.8889. Finally, the command of Gauss1D [3, 3] calls three point gauss rule and x_i and w_i values of third point and returns x_3 = +0.7746 and w_3 =0.5556 .

By using these solutions, a definite integral can be evaluated numerically. The formulization of the rule applied can be expressed as;

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=1}^{3} w_i f(x_i) = w_1 f(x_1) + w_2 f(x_2) + w_3 f(x_3) \tag{6.20}$$

The numerical values calculated from the Gauss1D module are sketched in Figure 6.4 while the areas of the circles are proportional to the w_i values.

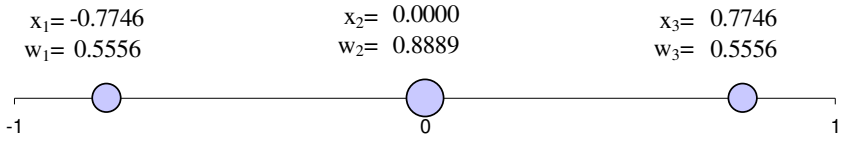


Figure 6.4. Schematic representation of 1D Gauss rule

Integration on square elements usually relies on tensor products of the one dimensional formulas. Since, the classical one dimensional Gauss integration rules are defined by,

$$\int_{-1}^{1} F(\xi)\, d\xi \;\approx\; \sum_{i=1}^{p} w_i F(\xi_i) \tag{6.21}$$

It can be integrated alternately;

$$\int_{-1}^{1}\int_{-1}^{1} F(\xi,\eta)\, d\xi\, d\eta = \int_{-1}^{1} d\eta = \int_{-1}^{1} F(\xi,\eta)\, d\xi \approx \sum_{i=1}^{p_1}\sum_{j=1}^{p_2} w_i w_j F(\xi_i,\eta_j) \tag{6.22}$$

where p1 and p2 are the number of Gauss points in the ξ and η directions, respectively.

6.3.3. Formal Parameters of Gauss2D:

The Java code shown in Figure 6.5 called "Gauss2D" returns ξ, η and $w_i \bullet w_j$ values over a straight-sided quadrilateral region.

NumPnt: Dimension of 2D Gauss integration rule varies from 1x1 to 5x5
{An integer number between 1 and 5}

Order: The point on which information is required (info values)
{An integer number between 1 and 25}

```
public static double[]gauss2D (int []NumPnt, int []Order)
   {
      int mp1,mp2;
      if(NumPnt.length==1)
         mp1=mp2=NumPnt[0];
      else{ mp1=NumPnt[0]; mp2=NumPnt[1];}
      if(mp1<0) ;
      int i,j;
      if(Order.length==1)
      {int m=Order[0];
         j=(int)Math.floor(((m-1.0)/mp1)+1);
         i=m-mp1*(j-1); }
      else
      {i=Order[0]; j=Order[1]; }
      double []r1=Gauss1D (mp1,i);
      double []r2=Gauss1D (mp2,j);
      double []info={r1[0],r2[0],r1[1]*r2[1]};
      return info;
   }
```

Figure 6.5. Gauss2D module

6.3.4. Example Commands for Gauss2D:

The command of Gauss2D [3, 4]

calls: two dimensional Gauss values of a "3×3" multiplication (mp1=mp2=3)
m= 4, j= 2, i= 1, r1= (-0.7746, 0.5556) and r2= (0, 0.8889)

returns: 4^{th} point coordinates(ξ_4 and η_4) and amplification value ($w_1 \bullet w_2$)
ξ_4= -0.7746, η_4= 0, $w_1 \bullet w_2$= 0.4938

The command of Gauss2D [3, 8]

calls: two dimensional Gauss values of a "3×3" multiplication (mp1=mp2=3)
m= 8, j= 2, i= 3, r1= (0, 0.8889), r2= (0.7746, 0.5556)

returns: 8^{th} point coordinates(ξ_8 and η_8) and amplification value ($w_2 \bullet w_3$)
ξ_8= 0, η_8= 0.7746, $w_2 \bullet w_3$= 0.4938

The command of Gauss2D [3, 9]

calls: two dimensional Gauss values of a "3×3" multiplication (mp1=mp2=3)
m= 9, j= 3, i= 3, r1= (0.7746, 0.5556), r2= (0.7746, 0.5556)

returns: 9^{th} point coordinates(ξ_9 and η_9) and amplification value ($w_3 \bullet w_3$)
ξ_9= 0.7746, η_9= 0.7746, $w_3 \bullet w_3$= 0.3086

The numerical values of ξ_b, η_b and ($w_i \bullet w_{j)}$ calculated from the Gauss2D module are sketched in Figure 6.4 while the areas of the circles are proportional to the $w_i \bullet w_j$ values.

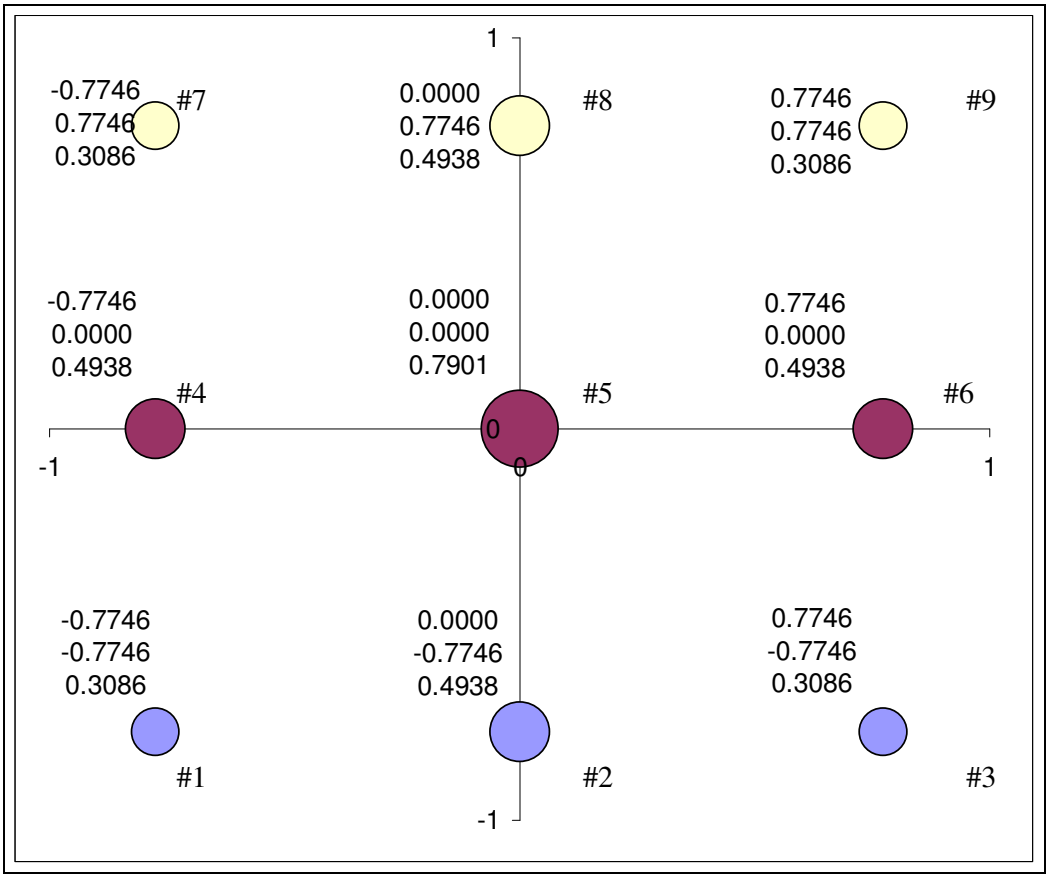


Figure 6.6. Schematic representation of 2D Gauss rule

6.4. Implementation of Stiffness Matrix of 4 Point Quadrilateral

The canonical form of the stiffness matrix of a quadrilateral element which is derived in equation 6.14 is repeated here for convenience:

$$[k_m] = \int_{-1}^{1} \int_{-1}^{1} F(\xi,\eta) d\xi d\eta \tag{6.23}$$

$$F(\xi,\eta) = [B]^T [D][B] \det[J] \tag{6.24}$$

Two dimensional Gauss quadrature rule states the following equation;

$$\int_{-1}^{1}\int_{-1}^{1} F(\xi,\eta)\, d\xi\, d\eta = \int_{-1}^{1} d\eta = \int_{-1}^{1} F(\xi,\eta)\, d\xi \approx \sum_{i=1}^{p_1}\sum_{j=1}^{p_2} w_i w_j F(\xi_i,\eta_j) \tag{6.25}$$

Applying two dimensional Gauss quadrature integration to the stiffness integration leads to the numerical solution of the stiffness matrix of a quadrilateral element. The Java code shown in Figure 6.8 called Pnt4QdrMembStiff returns global stiffness matrix of 4 point Quadrilateral element for plane stress which has 2 degrees of freedom at each nodes.

6.4.1. Formal Parameters of Pnt4QdrMembStiff

XYcoord:	Cartesian node coordinates of the quadrilateral element {(x1,y1),(x2,y2), (x3,y3),(x4,y4)}
ElasticMatrix:	The modulus of elasticity matrix of the plane stress element. {(E11, E12, E13), (E21, E22, E23), (E31,E32, E33)}
thickness:	Thickness of the quadrilateral element nodes {(th1, th2, th3, th4)}
p:	Dimension of 2D Gauss integration rule varies from 1x1 to 5x5. {An integer number between 1 and 5}

Pnt4QdrMembStiff module computes the stiffness matrix of the quadrilateral element sketched in Figure 6.7. The Pnt4QdrMembStiff module uses a local independent module called Pnt4QdrJacobi which accommodates the shape function values, shape function x and y derivatives and Jacobian determinant of the 4 point quadrilateral element.

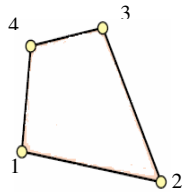


Figure 6.7. The 4 point quadrilateral element

Shape functions chosen for 4 point quadrilateral elements are directly implemented into the local variable Shapefnc. Main stiffness module substitutes equation (6.24) to equation (6.25) by using matrix multiplication modules which are introduced in MatrixOperations class. Finally, 8x8 stiffness matrix of the 4 point quadrilateral plane stress element is evaluated.

```
public static double[][] Pnt4QdrMembStiff(double [][]XYCoord,
double [][]ElasticMatrix, double []thickness,int p)
  { double []thick;
     if(thickness.length==1)
     {thick=new double[4];
        for(int i=0;i<thick.length;i++)
           thick[i]=thickness[0];}
     else
        thick=thickness;
     double [][]k=MatrixOperations.zero_matrix(8);double h,c;
     for(int i=0;i<p*p;i++)
     {
        int []rule={p};
        int []point={i+1};
        double []result1 = GaussRule.Gauss2D(rule,point);
        double []KsiNucoord=new double[2];
        KsiNucoord[0]=result1[0];
        KsiNucoord[1]=result1[1];
        double [][]result2=ShapeFunction.
       Pnt4QdrJacobi(XYCoord,KsiNucoord);
        h=MatrixOperations.dot_product(thick,result2[0]);
        c=result1[2]*result2[3][0]*h;
        double [][]BM=MatrixOperations.zero_matrix(3,8);
        for(int j=0;j<4;j++)
        {
       BM[0][j*2]=result2[1][j];     BM[1][j*2+1]=result2[2][j];
       BM[2][j*2+1]=result2[1][j]; BM[2][j*2]=result2[2][j];
        }
        k=MatrixOperations.add(k
       ,MatrixOperations.multiply_with_scalar(
        MatrixOperations.multiply(MatrixOperations.transpose(BM)
       ,(MatrixOperations.multiply(ElasticMatrix,BM))),c));
```

```
        }
        return k;
    }
 public static double[][] Pnt4QdrJacobi(double [][] XYcoord,
double [] KsiNucoord)
    {  double ksi=KsiNucoord[0];
        double nu=KsiNucoord[1];
        double []Shapefnc={(1-ksi)*(1-nu)/4.0,(1+ksi)*(1-nu)/4.0,
                           (1+ksi)*(1+nu)/4.0,(1-ksi)*(1+nu)/4.0};
        double []dNksi ={-(1-nu)/4.0, (1-nu)/4.0,(1+nu)/4.0,-(1+nu)/4.0};
        double []dNnu= {-(1-ksi)/4.0,-(1+ksi)/4.0,(1+ksi)/4.0,
                                                    (1-ksi)/4.0};
        double []x=new double[4];
        double []y=new double[4];
        for(int i=0;i<4;i++)
        {x[i]=XYcoord[i][0];y[i]=XYcoord[i][1];}
        double J11=MatrixOperations.dot_product(dNksi,x);
        double J12=MatrixOperations.dot_product(dNksi,y);
        double J21=MatrixOperations.dot_product(dNnu,x);
        double J22=MatrixOperations.dot_product(dNnu,y);
        double Jdet=J11*J22-J12*J21;
        double []dNx=MatrixOperations.multiply_with_scalar(
              (MatrixOperations.subtract(MatrixOperations.
multiply_with_scalar(dNksi,J22),
              MatrixOperations.multiply_with_scalar
(dNnu,J12))),1.0/Jdet);
        double []dNy=MatrixOperations.multiply_with_scalar(
              (MatrixOperations.add(MatrixOperations.
multiply_with_scalar(dNksi,-J21),
              MatrixOperations.multiply_with_scalar
(dNnu,J11))),1.0/Jdet);
        double [][]result=new double[4][];
        result[0]=new double[4];
        for(int i=0;i<4;i++)
           result[0][i]=Shapefnc[i];
        result[1]=new double[4];
        for(int i=0;i<4;i++)
           result[1][i]=dNx[i];
        result[2]=new double[4];
        for(int i=0;i<4;i++)
           result[2][i]=dNy[i];
        result[3]=new double[1];
        result[3][0]=Jdet;
        return result;
    }
```

Figure 6.8. Stiffness matrix of 4 point quadrilateral element

6.4.2. Example Commands for Pnt4QdrMembStiff

The stiffness matrix of the rectangular finite element sketched in Figure 6.9 is evaluated by Pnt4QdrMembStiff module.

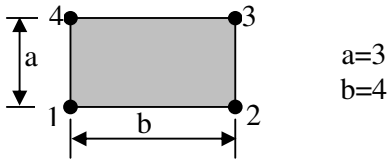


Figure 6.9. 4 point quadrilateral element example

The command of Pnt4QdrMembStiff [{(0,0),(4,0),(4,3),(0,3)}, {(100,40,0), (40,100,0), (0,0,30)}, {1,1,1,1}, {1}]

calls: node coordinates of a rectangular element

with elastic modules of $E = \begin{bmatrix} 100 & 40 & 0 \\ 40 & 100 & 0 \\ 0 & 0 & 30 \end{bmatrix}$

with constant thickness th=1

and two dimensional 1x1 gauss rule of 4 point quadrilateral element

returns: stiffness matrix of rectangular element Ke1x1

$$Ke1x1 = \begin{pmatrix} 28.75 & 17.5 & -8.75 & 2.5 & -28.75 & -17.5 & 8.75 & -2.5 \\ 17.5 & 38.9583 & -2.5 & 27.7083 & -17.5 & -38.9583 & 2.5 & -27.7083 \\ -8.75 & -2.5 & 28.75 & -17.5 & 8.75 & 2.5 & -28.75 & 17.5 \\ 2.5 & 27.7083 & -17.5 & 38.9583 & -2.5 & -27.7083 & 17.5 & -38.9583 \\ -28.75 & -17.5 & 8.75 & -2.5 & 28.75 & 17.5 & -8.75 & 2.5 \\ -17.5 & -38.9583 & 2.5 & -27.7083 & 17.5 & 38.9583 & -2.5 & 27.7083 \\ 8.75 & 2.5 & -28.75 & 17.5 & -8.75 & -2.5 & 28.75 & -17.5 \\ -2.5 & -27.7083 & 17.5 & -38.9583 & 2.5 & 27.7083 & -17.5 & 38.9583 \end{pmatrix} \quad (6.26)$$

The command of Pnt4QdrMembStiff [{(0,0),(4,0),(4,3),(0,3)}, {(100,40,0), (40,100,0), (0,0,30)}, {1,1,1,1}, {2}]

calls: two dimensional 2x2 gauss rule of the same element

returns: stiffness matrix of rectangular element Ke2x2

$$Ke2x2=\begin{pmatrix} 38.3333 & 17.5 & -18.3333 & 2.5 & -19.1667 & -17.5 & -0.833333 & -2.5 \\ 17.5 & 51.9444 & -2.5 & 14.7222 & -17.5 & -25.9722 & 2.5 & -40.6944 \\ -18.3333 & -2.5 & 38.3333 & -17.5 & -0.833333 & 2.5 & -19.1667 & 17.5 \\ 2.5 & 14.7222 & -17.5 & 51.9444 & -2.5 & -40.6944 & 17.5 & -25.9722 \\ -19.1667 & -17.5 & -0.833333 & -2.5 & 38.3333 & 17.5 & -18.3333 & 2.5 \\ -17.5 & -25.9722 & 2.5 & -40.6944 & 17.5 & 51.9444 & -2.5 & 14.7222 \\ -0.833333 & 2.5 & -19.1667 & 17.5 & -18.3333 & -2.5 & 38.3333 & -17.5 \\ -2.5 & -40.6944 & 17.5 & -25.9722 & 2.5 & 14.7222 & -17.5 & 51.9444 \end{pmatrix} \quad (6.27)$$

The command of Pnt4QdrMembStiff [{(0,0),(4,0),(4,3),(0,3)}, {(100,40,0), (40,100,0), (0,0,30)}, {1,1,1,1}, {3}]

calls: two dimensional 3x3 gauss rule of the same element

returns: stiffness matrix of rectangular element Ke3x3

$$Ke3x3=\begin{pmatrix} 38.3333 & 17.5 & -18.3333 & 2.5 & -19.1667 & -17.5 & -0.833333 & -2.5 \\ 17.5 & 51.9444 & -2.5 & 14.7222 & -17.5 & -25.9722 & 2.5 & -40.6944 \\ -18.3333 & -2.5 & 38.3333 & -17.5 & -0.833333 & 2.5 & -19.1667 & 17.5 \\ 2.5 & 14.7222 & -17.5 & 51.9444 & -2.5 & -40.6944 & 17.5 & -25.9722 \\ -19.1667 & -17.5 & -0.833333 & -2.5 & 38.3333 & 17.5 & -18.3333 & 2.5 \\ -17.5 & -25.9722 & 2.5 & -40.6944 & 17.5 & 51.9444 & -2.5 & 14.7222 \\ -0.833333 & 2.5 & -19.1667 & 17.5 & -18.3333 & -2.5 & 38.3333 & -17.5 \\ -2.5 & -40.6944 & 17.5 & -25.9722 & 2.5 & 14.7222 & -17.5 & 51.9444 \end{pmatrix} \quad (6.28)$$

Using gauss rule of 2x2 and rule 3x3 gives the same stiffness matrices, besides using 1x1 rule causes a rank deficient matrix. Since the entries of strain-displacement matrix vary linearly and Jacobian matrix is constant, 2x2 Gauss rule is capable to evaluate the integral exactly. The concept of the rank sufficiency of plane stress elements are expressed in Appendix A.

6.5. Implementation of Stiffness Matrix of 8 Point Quadrilateral

Since the procedure of constructing computer algebra is typical for all quadrilateral elements, the program modules of the higher order elements are evaluated systematically.

The Java code shown in Figure 6.11 called Pnt8QdrMembStiff returns global stiffness matrix of 8 point quadrilateral element for plane stress which has two degrees of freedom at each nodes.

6.5.1. Formal Parameters of Pnt8QdrMembStiff

XYcoord:	Cartesian node coordinates of the quadrilateral element {(x1,y1),(x2,y2),(x3,y3),(x4,y4),(x5,y5),(x6,y6),(x7,y7),(x8,y8)}
ElasticMatrix:	The modulus of elasticity matrix of the plane stress element. {(E11, E12, E13), (E21, E22, E23), (E31, E32, E33)}
thickness:	Thickness of the quadrilateral element nodes {(th1, th2, th3, th4, th5, th6, th7, th8)}
p:	Dimension of 2D Gauss integration rule {An integer number between 1 and 5}

Pnt8QdrMembStiff module computes the stiffness matrix of the serendipity quadrilateral element sketched in Figure 6.10. The Pnt8QdrMembStiff module uses a local independent module called Pnt8QdrJacobi which accommodates the shape function values, shape function x and y derivatives and Jacobian determinant of the 8 point quadrilateral element.

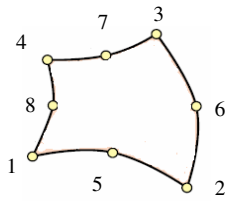


Figure 6.10. The 8 point quadrilateral element

Shape functions chosen for 8 point quadrilateral elements are directly implemented into the local variable Shapefnc. Main stiffness module substitutes equation (6.24) to equation (6.25) by using matrix multiplication modules which are introduced in MatrixOperations class. Finally, 16x16 stiffness matrix of the 8 point quadrilateral plane stress element is evaluated.

```
public static double[][] Pnt8QdrMembStiff(double [][]XYCoord,
double [][]ElasticMatrix, double []thickness,int p)
   {
      double []thick;
      if(thickness.length==1)
      {
         thick=new double[8];
         for(int i=0;i<thick.length;i++)
            thick[i]=thickness[0];
      }
      else
         thick=thickness;
      double [][]k=MatrixOperations.zero_matrix(16);
      double h,c;
      for(int i=0;i<p*p;i++)
      {
         int []rule={p};
         int []point={i+1};
         double []result1 = GaussRule.Gauss2D(rule,point);
         double []KsiNucoord=new double[2];
         KsiNucoord[0]=result1[0];
         KsiNucoord[1]=result1[1];
         double
[][]result2=ShapeFunction.Pnt8QdrJacobi(XYCoord,KsiNucoord);
         h=MatrixOperations.dot_product(thick,result2[0]);
         c=result1[2]*result2[3][0]*h;
         double [][]BM=MatrixOperations.zero_matrix(3,16);
         for(int j=0;j<8;j++)
         {
            BM[0][j*2]=result2[1][j];
            BM[1][j*2+1]=result2[2][j];
            BM[2][j*2+1]=result2[1][j];
            BM[2][j*2]=result2[2][j];
         }
         k=MatrixOperations.add(k
               ,MatrixOperations.multiply_with_scalar(
                  MatrixOperations.multiply(MatrixOperations.transpose(BM)
                     ,(MatrixOperations.multiply(ElasticMatrix,BM))),c));
      }
      return k;
   }
public static double[][] Pnt8QdrJacobi(double [][] XYcoord,double []
KsiNucoord)
   {
      double ksi=KsiNucoord[0];
      double nu=KsiNucoord[1];
      double []Shapefnc= {-(1-ksi)*(1-nu)*(1+ksi+nu)/4,
                          -(1+ksi)*(1-nu)*(1-ksi+nu)/4,
            -(1+ksi)*(1+nu)*(1-ksi-nu)/4,-(1-ksi)*(1+nu)*(1+ksi-nu)/4,
```

```
                (1+ksi)*(1-ksi)*(1-nu)/2, (1+ksi)*(1+nu)*(1-nu)/2,
                (1+ksi)*(1-ksi)*(1+nu)/2, (1-ksi)*(1+nu)*(1-nu)/2};

      double []dNksi={(1-nu)*(2*ksi+nu)/4,    (1-nu)*(2*ksi-nu)/4,
                           (1+nu)*(2*ksi+nu)/4,   (1+nu)*(2*ksi-nu)/4,
            ksi*(nu-1),  (1+nu)*(1-nu)/2,   -ksi*(1+nu), -(1+nu)*(1-nu)/2};
      double []dNnu={(1-ksi)*(ksi+2*nu)/4,   -(1+ksi)*(ksi-2*nu)/4,
                  (1+ksi)*(ksi+2*nu)/4,   -(1-ksi)*(ksi-2*nu)/4,
                 -(1+ksi)*(1-ksi)/2, -(1+ksi)*nu, (1+ksi)*(1-ksi)/2,-(1-ksi)*nu};
      double []x=new double[8];
      double []y=new double[8];
      for(int i=0;i<8;i++)
      {
         x[i]=XYcoord[i][0];
         y[i]=XYcoord[i][1];
      }
      double J11=MatrixOperations.dot_product(dNksi,x);
      double J21=MatrixOperations.dot_product(dNksi,y);
      double J12=MatrixOperations.dot_product(dNnu,x);
      double J22=MatrixOperations.dot_product(dNnu,y);
      double Jdet=J11*J22-J12*J21;
      double []dNx=MatrixOperations.multiply_with_scalar(

(MatrixOperations.subtract(MatrixOperations.multiply_with_scalar(dNksi,J22),
            MatrixOperations.multiply_with_scalar(dNnu,J21))),1.0/Jdet);
      double []dNy=MatrixOperations.multiply_with_scalar(

(MatrixOperations.add(MatrixOperations.multiply_with_scalar(dNksi,-J12),
            MatrixOperations.multiply_with_scalar(dNnu,J11))),1.0/Jdet);
      double [][]result=new double[4][];
      result[0]=new double[8];
      for(int i=0;i<8;i++)
         result[0][i]=Shapefnc[i];
      result[1]=new double[8];
      for(int i=0;i<8;i++)
         result[1][i]=dNx[i];
      result[2]=new double[8];
      for(int i=0;i<8;i++)
         result[2][i]=dNy[i];
      result[3]=new double[1];
      result[3][0]=Jdet;
      return result;
   }
```

Figure 6.11. Stiffness matrix of 8 point quadrilateral element

6.6. Implementation of Stiffness Matrix of 9 Point Quadrilateral

The Java code shown in Figure 6.13 called Pnt9QdrMembStiff returns global stiffness matrix of 9 point quadrilateral element for plane stress which has two degrees of freedom at each nodes.

6.6.1. Formal Parameters of Pnt9QdrMembStiff

XYcoord:	Cartesian node coordinates of the quadrilateral element {(x1,y1),(x2,y2),(x3,y3),(x4,y4),(x5,y5),(x6,y6),(x7,y7),(x8,y8), (x9,y9)}
ElasticMatrix:	The modulus of elasticity matrix of the plane stress element. {(E11, E12, E13), (E21, E22, E23), (E31,E32, E33)}
thickness:	Thickness of the quadrilateral element nodes {(th1, th2, th3, th4, th5, th6, th7, th8,th9)}
p:	Dimension of 2D Gauss integration rule {An integer number between 1 and 5}

Pnt9QdrMembStiff module computes the stiffness matrix of the serendipity quadrilateral element sketched in Figure 6.12. The Pnt9QdrMembStiff module uses a local independent module called Pnt9QdrJacobi which accommodates the shape function values, shape function x and y derivatives and Jacobian determinant of the 9 point quadrilateral element.

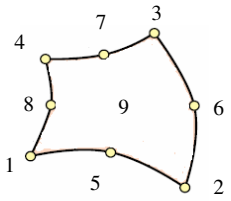


Figure 6.12. The 9 point quadrilateral element

Shape functions chosen for 9 point quadrilateral elements are directly implemented into the local variable Shapefnc. Main stiffness module substitutes equation (6.24) to equation (6.25) by using matrix multiplication modules which are introduced in

MatrixOperations class. Finally, 18x18 stiffness matrix of the 9 point quadrilateral plane stress element is evaluated.

```
public static double[][] Pnt9QdrMembStiff(double [][]XYCoord, double
[][]ElasticMatrix, double []thickness,int p)
   {
      double []thick;
      if(thickness.length==1)
      {
         thick=new double[9];
         for(int i=0;i<thick.length;i++)
            thick[i]=thickness[0];
      }
      else
         thick=thickness;
      double [][]k=MatrixOperations.zero_matrix(18);
      double h,c;
      for(int i=0;i<p*p;i++)
      {
         int []rule={p};
         int []point={i+1};
         double []result1 = GaussRule.Gauss2D(rule,point);
         double []KsiNucoord=new double[2];
         KsiNucoord[0]=result1[0];
         KsiNucoord[1]=result1[1];
         double
[][]result2=ShapeFunction.Pnt9QdrJacobi(XYCoord,KsiNucoord);
         h=MatrixOperations.dot_product(thick,result2[0]);
         c=result1[2]*result2[3][0]*h;
         double [][]BM=MatrixOperations.zero_matrix(3,18);
         for(int j=0;j<9;j++)
         {
            BM[0][j*2]=result2[1][j];
            BM[1][j*2+1]=result2[2][j];
            BM[2][j*2+1]=result2[1][j];
            BM[2][j*2]=result2[2][j];
         }
         k=MatrixOperations.add(k
               ,MatrixOperations.multiply_with_scalar(

MatrixOperations.multiply(MatrixOperations.transpose(BM)
                     ,(MatrixOperations.multiply(ElasticMatrix,BM))),c));
      }
      return k;
   }
public static double [][] Pnt9QdrJacobi(double [][]XYcoord, double
[]KsiNucoord)
   {
```

```
        double ksi=KsiNucoord[0];
        double nu=KsiNucoord[1];
        double []Shapefnc={(1-ksi)*ksi*(1-nu)*nu/4,
                  -(1+ksi)*ksi*(1-nu)*nu/4,
                  (1+ksi)*ksi*(1+nu)*nu/4,-(1-ksi)*ksi*(1+nu)*nu/4,
                 -(1-ksi*ksi)*(1-nu)*nu/2, (1+ksi)*ksi*(1-nu*nu)/2,
                  (1-ksi*ksi)*(1+nu)*nu/2,-(1-ksi)*ksi*(1-nu*nu)/2,
                  (1-ksi*ksi)*(1-nu*nu)};
        double []dNksi={(1-2*ksi)*nu*(1-nu)/4,-(1+2*ksi)*nu*(1-nu)/4,
              (1+2*ksi)*nu*(1+nu)/4,-(1-2*ksi)*nu*(1+nu)/4,
               ksi*nu*(1-nu), (1.0/2+ksi)*(1-nu*nu),
              -ksi*nu*(1+nu),-(1.0/2-ksi)*(1-nu*nu), 2*ksi*(nu*nu-1)};
        double []dNnu={ksi*(1-ksi)*(1-2*nu)/4, -ksi*(1+ksi)*(1-2*nu)/4,
              ksi*(1+ksi)*(1+2*nu)/4, -ksi*(1-ksi)*(1+2*nu)/4,
              -(1.0/2-nu)*(1-ksi*ksi),  -ksi*(1+ksi)*nu,
               (1.0/2+nu)*(1-ksi*ksi),   ksi*(1-ksi)*nu,2*(ksi*ksi-1)*nu};
        double []x=new double[9];
        double []y=new double[9];
        for(int i=0;i<9;i++)
        {
           x[i]=XYcoord[i][0];
           y[i]=XYcoord[i][1];
        }
        double J11=MatrixOperations.dot_product(dNksi,x);
        double J21=MatrixOperations.dot_product(dNksi,y);
        double J12=MatrixOperations.dot_product(dNnu,x);
        double J22=MatrixOperations.dot_product(dNnu,y);
        double Jdet=J11*J22-J12*J21;
        double []dNx=MatrixOperations.multiply_with_scalar(

MatrixOperations.add(MatrixOperations.multiply_with_scalar(dNksi,J22),
                  MatrixOperations.multiply_with_scalar(dNnu,-
J21)),1.0/Jdet);
        double []dNy=MatrixOperations.multiply_with_scalar(

MatrixOperations.add(MatrixOperations.multiply_with_scalar(dNksi,-
J12),
                  MatrixOperations.multiply_with_scalar(dNnu,J11)),1.0/Jdet);
        double [][]result=new double[4][];
        result[0]=Shapefnc;
        result[1]=dNx;
        result[2]=dNy;
        result[3]=new double[1];
        result[3][0]=Jdet;
        return result;
    }
```

Figure 6.13. Stiffness matrix of 9 point quadrilateral element

7. JAVA IMPLEMANTATION OF TRIANGULAR ELEMENTS

7.1. Introduction

The constant stress triangle was the first element, which was developed for finite element analysis and it is still used in some finite element programs, which are being marketed worldwide. A triangle is one of the basic shapes of geometry: a polygon with three corners or vertices and three sides or edges which are straight line segments.

In order to implement stiffness matrix of a general constant triangular element which is given by the equation (2.94), shape function matrix [N] is need to be defined. The generic displacements at any point inside a triangular element in terms of nodal displacements of the element are defined by using the area coordinates. The Cartesian and the area coordinated of a typical triangle are shown in Figure 7.1.

In figure 7.1,for a point P within the domain of the triangle, the area coordinates are

$$P(\xi_1, \eta_2, \zeta_3) \tag{7.1}$$

where

$$\begin{aligned} \xi_1 &= \frac{A_1}{A} = \frac{h_1}{H_1} \\ \eta_2 &= \frac{A_2}{A} \\ \zeta_3 &= \frac{A_3}{A} \\ \xi_1 + \eta_2 + \zeta_3 &= 1 \end{aligned} \tag{7.2}$$

Cartesian and area coordinates are linked by the relation;

$$\begin{aligned} x &= a_1\xi_1 + a_2\eta_2 + a_3\zeta_3 \\ y &= b_1\xi_1 + b_2\eta_2 + b_3\zeta_3 \end{aligned} \tag{7.3}$$

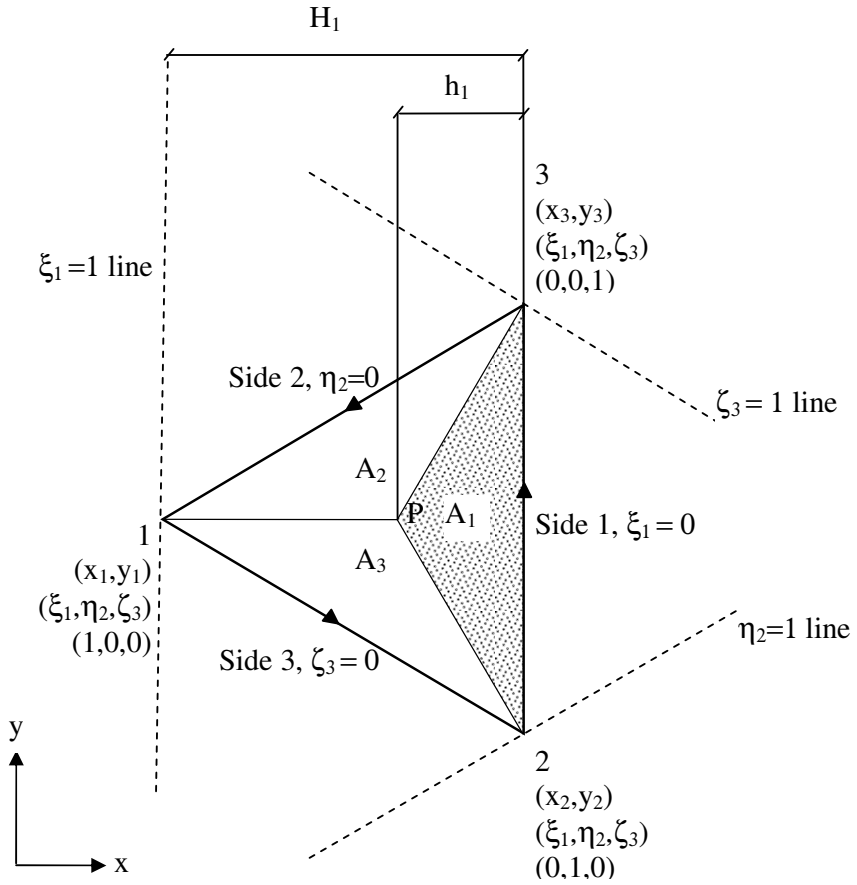


Figure 7.1. Area coordinates of a triangle

Since the values of the area coordinate are known at the coordinates of the corners, the unknown constants can be derived from equation 7.3. Thus, solutions obtained are

$$\begin{aligned} a_1 &= x_1, b_1 = y_1 \\ a_2 &= x_2, b_2 = y_2 \\ a_3 &= x_3, b_3 = y_3 \end{aligned} \tag{7.4}$$

Therefore, the relationship can be written in matrix form as;

$$\begin{bmatrix} 1 \\ x \\ y \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{bmatrix} \begin{bmatrix} \xi_1 \\ \eta_2 \\ \zeta_3 \end{bmatrix} \tag{7.5}$$

By inverting equation (7.5), area coordinates can be obtained in terms of Cartesian coordinates.

$$\begin{bmatrix} \xi_1 \\ \eta_2 \\ \zeta_3 \end{bmatrix} = \frac{1}{2A} \begin{bmatrix} (x_2 y_3 - x_3 y_2) & (y_2 - y_3) & (x_3 - x_2) \\ (x_3 y_1 - x_1 y_3) & (y_3 - y_1) & (x_1 - x_3) \\ (x_1 y_2 - x_2 y_1) & (y_1 - y_2) & (x_3 - x_1) \end{bmatrix} \begin{bmatrix} 1 \\ x \\ y \end{bmatrix} \tag{7.6}$$

By applying chain rule to equation 7.6, partial derivative of area coordinates with respect to x and y can be calculated as follows

$$\begin{aligned} \frac{\partial f}{\partial x} &= \frac{1}{2A}\left(\frac{\partial f}{\partial \xi_1} y_{23} + \frac{\partial f}{\partial \eta_2} y_{31} + \frac{\partial f}{\partial \zeta_3} y_{12} \right) \\ \frac{\partial f}{\partial y} &= \frac{1}{2A}\left(\frac{\partial f}{\partial \xi_1} x_{32} + \frac{\partial f}{\partial \eta_2} x_{13} + \frac{\partial f}{\partial \zeta_3} x_{21} \right) \end{aligned} \tag{7.7}$$

which in the matrix form:

$$\begin{bmatrix} \dfrac{\partial f}{\partial x} \\ \dfrac{\partial f}{\partial y} \end{bmatrix} = \frac{1}{2A} \begin{bmatrix} y_{23} & y_{31} & y_{12} \\ x_{32} & x_{13} & x_{21} \end{bmatrix} \begin{bmatrix} \dfrac{\partial f}{\partial \xi_1} \\ \dfrac{\partial f}{\partial \eta_2} \\ \dfrac{\partial f}{\partial \zeta_3} \end{bmatrix} \tag{7.8}$$

7.2. Implementation of Stiffness Matrix of Constant Stress Triangle

The element stiffness matrix for unit thickness is evaluated in equation (2.94) is repeated her for convenience:

$$[k_m] = \iint [B]^T [D][B]\, dxdy \tag{7.9}$$

Strain-displacement matrix, $[B]$ can be derived by using equation (7.8) and equation (2.93).

$$[B]=\frac{1}{2A}\begin{bmatrix} y_2-y_3 & 0 & y_3-y_1 & 0 & y_1-y_2 & 0 \\ 0 & x_3-x_2 & 0 & x_1-x_3 & 0 & x_2-x_1 \\ x_3-x_2 & y_2-y_3 & x_1-x_3 & y_3-y_1 & x_2-x_1 & y_1-y_2 \end{bmatrix} \tag{7.10}$$

and

$$[D]=\begin{bmatrix} E_{11} & E_{12} & E_{13} \\ E_{12} & E_{22} & E_{23} \\ E_{13} & E_{23} & E_{33} \end{bmatrix} \tag{7.11}$$

Since $[B]$ and $[D]$ are constant, the stiffness equation over triangular domain Ψ can be written as;

$$[k_m]=[B]^T[D][B]\int_{\Psi} h d\Psi \tag{7.12}$$

Here h is the nodal thickness of the triangular element. The implementation of constant stress triangle is obtained by substituting the strain-stress matrix and modulus of elasticity matrix in equation (7.12).

7.2.1. Formal Parameters of Pnt3TrgMembStiff

XYCoord: Cartesian node coordinates of the quadrilateral element.
{(x1,y1),(x2,y2), (x3,y3),(x4,y4)}

ElasticMatrix: The modulus of elasticity matrix of the plane stress element.
{(E11, E12, E13), (E21, E22, E23), (E31, E32, E33)}

thickness: Thickness of the quadrilateral element nodes
{(th1, th2, th3, th4)}

The Java code shown in Figure 7.2 called Pnt3TrgMembStiff computes global stiffness matrix of 3 point constant stress triangle which has 2 degrees of freedom at each nodes.

```
public static double[][] Pnt3TrgMembStiff(double [][]XYCoord,
double [][]ElasticMatrix,double []thickness)
   {double []th;
      if(thickness.length==1)
      { th=new double[3];
         for(int i=0;i<th.length;i++)
            th[i]=thickness[0]; }
      else th=thickness;
      double h=(th[0]+th[1]+th[2])/3.0;
      double x1=XYCoord[0][0];double y1=XYCoord[0][1];
      double x2=XYCoord[1][0]; double y2=XYCoord[1][1];
      double x3=XYCoord[2][0]; y3=XYCoord[2][1];
      double x21=x2-x1; double x13=x1-x3;
      double x32=x3-x2; double y12=y1-y2;
      double y31=y3-y1; double y23=y2-y3;
      double [][]BM={{y23,0,y31,0,y12,0},{0,x32,0,x13,0,x21},
          {x32,y23,x13,y31,x21,y12}};
      double A=(x21*y31-x13*y12)/2;
      BM=MatrixOperations.multiply_with_scalar(BM,1.0/(2*A));
      double [][]Ke=MatrixOperations.multiply_with_scalar
(MatrixOperations.multiply(MatrixOperations.transpose(BM)
        ,MatrixOperations.multiply(ElasticMatrix,BM)),A*h);
      return Ke; }
```

Figure 7.2. Pnt3TrgMembStiff module

7.3. Implementation of Gauss Rule for Triangular Elements

The two dimensional gauss quadrature rule for solving the stiffness equation of the two dimensional quadrilateral elements could not be applied to the triangle elements because of the requirements of the triangular geometry. Therefore, a new Gauss rule will be introduced to approximate the numerical values of the triangular element domain.

The simplest Triangular gauss rule has one sample point located at the centroid of the triangular element. For a straight sided triangle the area coordinates of the one point rule is given by the formula [17];

$$\frac{1}{A}\int_{\Psi} F(\xi_1,\eta_2,\zeta_3)d\Psi \approx F(\frac{1}{3},\frac{1}{3},\frac{1}{3}) \tag{7.13}$$

where Ψ is the triangular domain and A is the triangle area which is given in equation (7.14).

$$A = \int_{\Psi} d\Psi = \frac{1}{2}\begin{vmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \end{vmatrix} = \frac{1}{2}\left[(x_2 y_3 - x_3 y_2) + (x_3 y_1 - x_1 y_3) + (x_1 y_2 - x_2 y_1)\right] \qquad (7.14)$$

This rule is named p=1 in depicted in Figure 7.3.

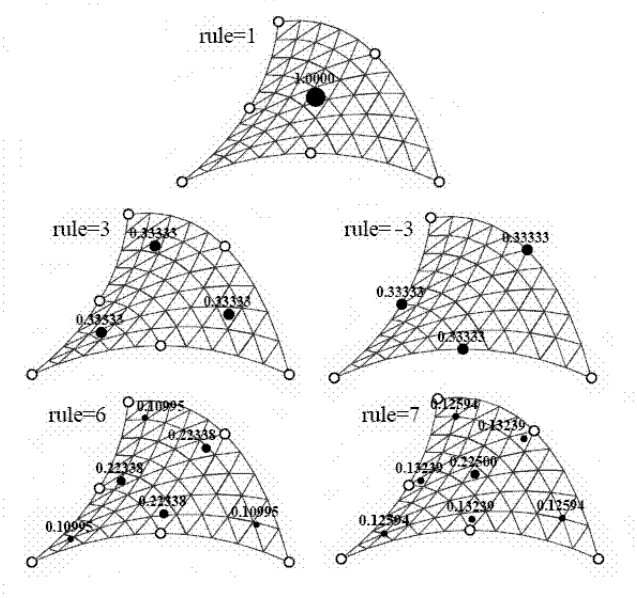


Figure 7.3. Triangular Gauss points

Three point rules are separated into two groups. Firstly, three sample points are taken at the midpoints of the triangular element sides. This rule is defined by P=-3, and it is calculated by the following equation.

$$\frac{1}{A}\int_{\Psi} F(\xi_1, \eta_2, \zeta_3) d\Psi \approx \frac{1}{3} F(\frac{1}{2}, \frac{1}{2}, 0) + \frac{1}{3} F(0, \frac{1}{2}, \frac{1}{2}) + \frac{1}{3} F(\frac{1}{2}, 0, \frac{1}{2}) \qquad (7.15)$$

Secondly, three points with equal weight are chosen interior of the domain. This rule is defined by P=3 and it is given by the following equation,

$$\frac{1}{A}\int_{\Psi} F(\xi_1, \eta_2, \zeta_3) d\Psi \approx \frac{1}{3} F(\frac{2}{3}, \frac{1}{6}, \frac{1}{6}) + \frac{1}{3} F(\frac{1}{6}, \frac{2}{3}, \frac{1}{6}) + \frac{1}{3} F(\frac{1}{6}, \frac{1}{6}, \frac{2}{3}) \qquad (7.16)$$

The values of the 6 and 7 point triangular Gauss rule which are depicted in Figure 7.3 are given by G.Strang & G.Fix. [17]. Gauss rule coefficients are implemented directly into the TriangularGauss module shown in Figure 7.4.

7.3.1. Formal Parameters of TriangularGauss

NumPnt: The rule associate with the triangular gauss rule (p)

{An integer number (-3, 1, 3, 6, 7)}

order: The point on which information is required

{An integer number between 1 and 7}

```
public static double [] TriangularGauss(int NumPnt, int Order)
   {double zeta,cnt1,cnt2; double p=NumPnt; int i=Order;
      if(p==1) {double []alfa={1.0/3,1.0/3,1.0/3,1};return alfa;}
      if(p==3) {double []alfa={1.0/6,1.0/6,1.0/6,1.0/3};
         alfa[i-1]=2.0/3; return alfa; }
      if(p==-3){ double []alfa={1.0/2,1.0/2,1.0/2,1.0/3};
         alfa[i-1]=0; alfa; }
      if(p==6)
{cnt1=(8Math.sqrt(10)+Math.sqrt(3844*Math.sqrt(2.0/5)))/18;
cnt2=(8-Math.sqrt(10)-Math.sqrt(38-44*Math.sqrt(2.0/5)))/18;

         if(i<4){
            double[] alfa={cnt1,cnt1,cnt1,(620+Math.sqrt(213125-
53320*Math.sqrt(10)))/3720};
            alfa[i-1]=1-2*cnt1; return alfa; }else if(i>3)
         {double[] alfa={cnt2,cnt2,cnt2,(620-Math.sqrt(213125-
53320*Math.sqrt(10)))/3720};alfa[i-3-1]=1-2*cnt2; return alfa;} }

    if(p==-6) {if(i<4)
         {double[] alfa={1.0/6,1.0/6,1.0/6,3.0/10};alfa[i-1]=2.0/3;
            return alfa; }
         else if(i>3) {double[] alfa={1.0/2,1.0/2,1.0/2,1.0/30};
            alfa[i-3-1]=0; return alfa; }}
    if(p==7) {cnt1=(6-Math.sqrt(15))/21;cnt2=(6+Math.sqrt(15))/21;
         if(i<4) {
            double []alfa={cnt1,cnt1,cnt1,(155-Math.sqrt(15))/1200};
            alfa[i-1]=1-2*cnt1; return alfa; }
         else if(i>3&&i<7)
         {double []alfa={cnt2,cnt2,cnt2,(155+Math.sqrt(15))/1200};
            alfa[i-3-1]=1-2*cnt2; return alfa; }
         else if(i==7) {double []alfa={1.0/3,1.0/3,1.0/3,9.0/40};
            return alfa;
         }}
      return null;
   }
```

Figure 7.4. Triangular Gauss module

7.3.2. Example Command for TriangularGauss

The command of TriangularGauss [3, 2]

calls: midpoint triangular gauss rule (P=3)

returns: triangular coordinates of the second point and amplification value

$$\xi_1 = \frac{1}{6},\ \eta_2 = \frac{2}{3},\ \zeta_3 = \frac{1}{6},\ w = 0.3333$$

7.4. Implementation of Stiffness Matrix of 6 Point Isoparametric Triangle

Element geometry of isoparametric elements is defined by the node coordinates and the relationship between Cartesian coordinates and natural coordinates derived in equation (6.4), which is repeated here:

$$x = \sum_{i=1}^{n} x_i N_i^e,\ y = \sum_{i=1}^{n} y_i N_i^e \tag{7.17}$$

where the shape functions and their derivatives are given as;

$$N^T = \begin{bmatrix} N_1^e \\ N_2^e \\ N_3^e \\ N_4^e \\ N_5^e \\ N_6^e \end{bmatrix} = \begin{bmatrix} \xi_1(2\xi_1 - 1) \\ \eta_2(2\eta_2 - 1) \\ \zeta_3(2\zeta_3 - 1) \\ 4\xi_1\eta_2 \\ 4\eta_2\zeta_3 \\ 4\zeta_3\xi_1 \end{bmatrix}$$

$$\frac{\partial \mathrm{N}^T}{\partial \xi_1} = \begin{bmatrix} 4\xi_1 - 1 \\ 0 \\ 0 \\ 4\eta_2 \\ 0 \\ 4\zeta_3 \end{bmatrix}, \frac{\partial \mathrm{N}^T}{\partial \eta_2} = \begin{bmatrix} 0 \\ 4\eta_2 - 1 \\ 0 \\ 4\xi_1 \\ 4\zeta_3 \\ 0 \end{bmatrix}, \frac{\partial \mathrm{N}^T}{\partial \zeta_3} = \begin{bmatrix} 0 \\ 0 \\ 4\zeta_3 - 1 \\ 0 \\ 4\eta_2 \\ 4\xi_1 \end{bmatrix} \tag{7.18}$$

Triangular coordinate partial derivatives can be evaluated by applying chain rule.

$$\begin{bmatrix} 1 & 1 & 1 \\ \sum x_i \frac{\partial N_i}{\partial \xi_1} & \sum x_i \frac{\partial N_i}{\partial \eta_2} & \sum x_i \frac{\partial N_i}{\partial \zeta_3} \\ \sum y_i \frac{\partial N_i}{\partial \xi_1} & \sum y_i \frac{\partial N_i}{\partial \eta_2} & \sum y_i \frac{\partial N_i}{\partial \zeta_3} \end{bmatrix} \begin{bmatrix} \frac{\partial \xi_1}{\partial x} & \frac{\partial \xi_1}{\partial y} \\ \frac{\partial \eta_2}{\partial x} & \frac{\partial \eta_2}{\partial y} \\ \frac{\partial \zeta_3}{\partial x} & \frac{\partial \zeta_3}{\partial y} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{7.19}$$

Equation 7.19 can be rewritten by taking into consideration of the Jacobian matrix introduce in equation (6.5).

$$\text{J P} = \begin{bmatrix} 1 & 1 & 1 \\ J_{x1} & J_{x2} & J_{x3} \\ J_{y1} & J_{y2} & J_{y3} \end{bmatrix} \begin{bmatrix} \frac{\partial \xi_1}{\partial x} & \frac{\partial \xi_1}{\partial y} \\ \frac{\partial \eta_2}{\partial x} & \frac{\partial \eta_2}{\partial y} \\ \frac{\partial \zeta_3}{\partial x} & \frac{\partial \zeta_3}{\partial y} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{7.20}$$

By using Jacobian transformations, the shape function derivatives can be found as;

$$\begin{aligned} \frac{\partial N_i}{\partial x} &= \frac{1}{2J}\left(\frac{\partial N_i}{\partial \xi_1} J_{y23} + \frac{\partial N_i}{\partial \eta_2} J_{y31} + \frac{\partial N_i}{\partial \zeta_3} J_{y12}\right) \\ \frac{\partial N_i}{\partial y} &= \frac{1}{2J}\left(\frac{\partial N_i}{\partial \xi_1} J_{x32} + \frac{\partial N_i}{\partial \eta_2} J_{x13} + \frac{\partial N_i}{\partial \zeta_3} J_{x21}\right) \end{aligned} \tag{7.21}$$

7.4.1. Formal Parameters of Pnt6TrgMembStiff

The Java code shown in Figure 7.6 called Pnt6TrgMembStiff returns global stiffness matrix of the 6 point isoparametric triangle element for plane stress which has 2 degrees of freedom at each nodes.

XYcoord:	Cartesian node coordinates of the quadrilateral element {(x1,y1),(x2,y2),(x3,y3),(x4,y4),(x5,y5),(x6,y6)}
ElasticMatrix:	The modulus of elasticity matrix of the plane stress element. {(E11, E12, E13), (E21, E22, E23), (E31, E32, E33)}
thickness:	Thickness of the quadrilateral element nodes

{(th1, th2, th3, th4, th5, th6,)}

p: Triangular Gauss integration rule

{An integer number (-3, 1, 3, 6, 7)}

Pnt6TrgMembStiff module computes the stiffness matrix of the 6 point isoparametric Triangle element sketched in Figure 7.5. The Pnt6TrgMembStiff module uses a local independent module called Pnt6TrgJacobi which accommodates the shape function values, shape function x and y derivatives and Jacobian determinant of the 6 point isoparametric triangle element.

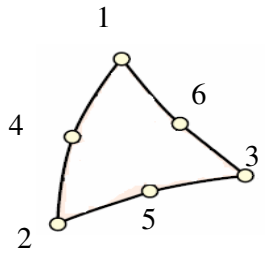


Figure 7.5. The 6 point isoparametric triangle element

Shape functions chosen for 6 point isoparametric triangle elements are directly implemented into the local variable Shapefnc. Main stiffness module substitutes equation (6.24) to equation (6.25) by using matrix multiplication modules which are introduced in MatrixOperations class. Finally, 12x12 stiffness matrix of the 6 point triangular plane stress element is evaluated.

Moreover, Java code shown in Figure 7.7, evaluates the shape function derivatives of the six node triangular element.

```
public static double [][] Pnt6TrgMembStiff(double
[][]XYCoord,double [][]ElasticMatrix,double []thickness,int p)
   {
      double []thick;
      if(thickness.length==1)
      {
         thick=new double[6];
         for(int i=0;i<thick.length;i++)
            thick[i]=thickness[0];
      }
      else
         thick=thickness;
      double [][]Ke=MatrixOperations.zero_matrix(12);
      for(int i=0;i<(int)Math.abs(p);i++)
      {
         double []result1=GaussRule.TriangularGauss(p,i+1);
         double []areacoor={result1[0],result1[1],result1[2]};
         double
[][]result2=ShapeFunction.Pnt6TrgJacobi(XYCoord,areacoor);
         double h=MatrixOperations.dot_product(thick,result2[0]);
         double c=result1[3]*result2[3][0]*h/2;
         double [][]BM=MatrixOperations.zero_matrix(3,12);
         for(int j=0;j<6;j++)
         {
            BM[0][j*2]=result2[1][j];
            BM[1][j*2+1]=result2[2][j];
            BM[2][j*2+1]=result2[1][j];
            BM[2][j*2]=result2[2][j];
         }
         Ke=MatrixOperations.add(Ke
               ,MatrixOperations.multiply_with_scalar(

MatrixOperations.multiply(MatrixOperations.transpose(BM)

,(MatrixOperations.multiply(ElasticMatrix,BM))),c));
      }
      return Ke;
   }
```

Figure 7.6. Pnt6TrgMembStiff module

```
public static double[][] Pnt6TrgJacobi(double [][]XYcoord, double
[]areacoor)
   {
      double ksi1=areacoor[0],nu2=areacoor[1],zeta3=areacoor[2];
double x1=XYcoord[0][0],x2=XYcoord[1][0],x3=XYcoord[2][0],
x4=XYcoord[3][0],x5=XYcoord[4][0],x6=XYcoord[5][0];
      double
y1=XYcoord[0][1],y2=XYcoord[1][1],y3=XYcoord[2][1],
y4=XYcoord[3][1],y5=XYcoord[4][1],y6=XYcoord[5][1];
      double
dx4,dx5,dx6,dy4,dy5,dy6,Jx21,Jx32,Jx13,Jy12,Jy23,Jy31;
      dx4=XYcoord[3][0]-(XYcoord[0][0]+XYcoord[1][0])/2;
      dx5=XYcoord[4][0]-(XYcoord[1][0]+XYcoord[2][0])/2;
      dx6=XYcoord[5][0]-(XYcoord[0][0]+XYcoord[2][0])/2;
      dy4=XYcoord[3][1]-(XYcoord[0][1]+XYcoord[1][1])/2;
      dy5=XYcoord[4][1]-(XYcoord[1][1]+XYcoord[2][1])/2;
      dy6=XYcoord[5][1]-(XYcoord[0][1]+XYcoord[2][1])/2;
      double []Shapefnc={ksi1*(2*ksi1-1),nu2*(2*nu2-
1),zeta3*(2*zeta3-1),4*ksi1*nu2,4*nu2*zeta3,4*zeta3*ksi1};
      Jx21= x2-x1+4*(dx4*(ksi1-nu2)+(dx5-dx6)*zeta3);
      Jx32= x3-x2+4*(dx5*(nu2-zeta3)+(dx6-dx4)*ksi1);
      Jx13= x1-x3+4*(dx6*(zeta3-ksi1)+(dx4-dx5)*nu2);
      Jy12= y1-y2+4*(dy4*(nu2-ksi1)+(dy6-dy5)*zeta3);
      Jy23= y2-y3+4*(dy5*(zeta3-nu2)+(dy4-dy6)*ksi1);
      Jy31= y3-y1+4*(dy6*(ksi1-zeta3)+(dy5-dy4)*nu2);
      double Jdet = Jx21*Jy31-Jy12*Jx13;
      double []dNx= {(4*ksi1-1)*Jy23,(4*nu2-1)*Jy31,(4*zeta3-
1)*Jy12,4*(nu2*Jy23+ksi1*Jy31),
         4*(zeta3*Jy31+nu2*Jy12),4*(ksi1*Jy12+zeta3*Jy23)};
      dNx=MatrixOperations.multiply_with_scalar(dNx,1.0/Jdet);
      double []dNy= {(4*ksi1-1)*Jx32,(4*nu2-1)*Jx13,(4*zeta3-
1)*Jx21,4*(nu2*Jx32+ksi1*Jx13),
         4*(zeta3*Jx13+nu2*Jx21),4*(ksi1*Jx21+zeta3*Jx32)};
      dNy=MatrixOperations.multiply_with_scalar(dNy,1.0/Jdet);
      double [][]result=new double[4][];
      result[0]=Shapefnc;
      result[1]=dNx;
      result[2]=dNy;
      result[3]=new double[1];
      result[3][0]=Jdet;
      return result;
   }
```

Figure 7.7. Pnt6TrgJacobi module

8. JAVA PROGRAM FOR PLANE STRESS PROBLEMS

8.1. Introduction

A class of common engineering problems has one principal stress that is much smaller than the other two. By assuming that this small principal stress is zero, the three-dimensional stress state can be reduced to two dimensions. Since the remaining two principal stresses lie in a plane, these simplified two dimensional problems are called plane stress problems.

Generally, members which has a small z dimension compared to the in-plane x and y dimensions and loads act only in the x-y plane can be considered to be under plane stress. Therefore, engineering problems such as thin plates with holes that are loaded in their plane resulting in local stress concentrations can be defined as a plane stress problem. Two examples of plane stress problems are depicted in Figure 8.1.

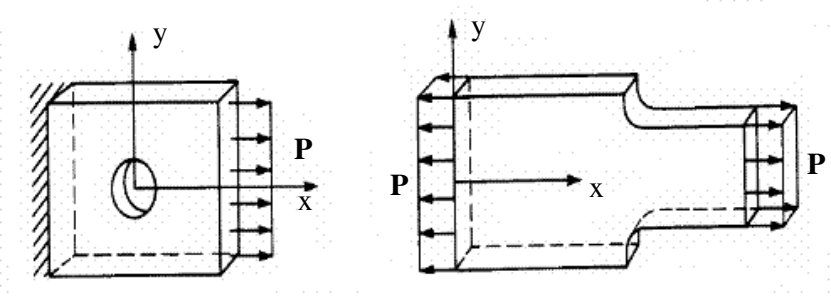


Figure 8.1. Examples of plane stress problems

The four major analysis stages, " definition, execution, solution and computation", introduced in 4.1 are also valid for Plane Stress program.

8.2. Implementation of Global Structural Stiffness Matrix

The Java code shown in Figure 8.2 called PSGlobStrStiff returns global structural matrix of the plane stress problems. Global structural stiffness matrix which accommodates

all displacement force relations of every degree of freedoms, is a square matrix with $\Sigma n_{freedom}$ dimensions. Total number of freedom can be calculated by equation (8.1).

$$\Sigma n_{freedom} = \text{total number of freedom} = 2^* \text{ number of nodes} \quad (8.1)$$

The plane stress elements which are evaluated previously are assigned to the numerical values that are given in Table 8.1. In order to obtain the convergence of the numerical solution, the required rank sufficient gauss rules in element level are also given in table 8.1.

Appendix A expresses the convergence requirements for the element level.

Table 8.1. Element classification

Element Name	Rank Sufficient Gauss Rule	Assigned Number for Input File
Pnt4QdrMembStiff	2x2	40
Pnt8QdrMembStiff	3x3	42
Pnt9QdrMembStiff	4x4	44
Pnt3TrgMembStiff	Triangular r=1 Rule	30
Pnt6TrgMembStiff	Triangular r=3 Rule	31
Pnt6TrgMembStiff	Triangular r=-3 Rule	32

8.2.1. Formal Parameters of PSGlobStrStiff

XYcoord: All global coordinates of element nodes
$\{(x_1, y_1), (x_2, y_2).....(x_{last}, y_{last})\}$

ElasticMatrix: The modulus of elasticity matrix of the plane stress elements.
{(E11, E12, E13), (E21, E22, E23), (E31,E32, E33)}

elempositions: Element end points that are assigned for the system joints.
$\{(\#1, \#2, \#3,\#4)_1, (\#1, \#2)_2..... (\#1, \#2)_{last}\}$

elemclass: Element class defined in Table 8.1 in the same order of "elemposition" parameter.
Related numbers are indicated in Table 8.1.

Thickness: Thickness of the each element in the same order of "elemposition" parameter $(th_1,th_{2,}....th_{last})$

```
public static double[][] PSGlobStrStiff(double [][]XYZCoord,
  double [][][]ElasticMatrix, int [][] elempositions, int []elemclass ,
double [][]Thickness)
   {int numberofnodes=XYZCoord.length;
      int numberofele=elempositions.length;
      int []OKtyp={40,42,44,30,31,32};
      int []OKenl= {4,8,9,3,6,6};
      double
[][]K=MatrixOperations.zero_matrix(2*numberofnodes);
      for(int i=0;i<numberofele;i++)
      {  int type=elemclass[i];
         int []enl=elempositions[i];
         int n=enl.length;
         boolean found=false;
         for(int j=0;j<OKtyp.length;j++)
         { if(OKtyp[j]==type)
            {  found=trelemdisp;
               if(OKenl[j]!=n)
                  return null;
               break;}}
         if(!found)//type can not be found
            return null;
         int []ongfi=new int[n*2];
          for(int j=0;j<n;j++)
           {  ongfi[j*2+0]=2*enl[j]-1;
              ongfi[j*2+1]=2*enl[j];
            }
         double [][]XYCoord=new double[n][];
         for(int j=0;j<n;j++)
         {
            XYCoord[j]=new double[XYZCoord[enl[j]-1].length];
            for(int k=0;k<XYCoord[j].length;k++)
               XYCoord[j][k]=XYZCoord[enl[j]-1][k];
         }
         double [][]Ke=null;
         if(type==40){double [][]Eelem=ElasticMatrix[i];
            double []th=Thickness[i];
            Ke=Pnt4QdrMembStiff(XYCoord,Eelem,th,2);}
         else if(type==42){double [][]Eelem=ElasticMatrix[i];
            double []th=Thickness[i];
            Ke=Pnt8QdrMembStiff(XYCoord,Eelem,th,3);}
         else if(type==44){double [][]Eelem=ElasticMatrix[i];
            double []th=Thickness[i];
            Ke=Pnt9QdrMembStiff(XYCoord,Eelem,th,4);}
         else if(type==30){double [][]Eelem=ElasticMatrix[i];
            double []th=Thickness[i];
            Ke=Pnt3TrgMembStiff(XYCoord,Eelem,th);}
         else if(type==31){ double [][]Eelem=ElasticMatrix[i];
            double []th=Thickness[i];
```

```
            Ke=Pnt6TrgMembStiff(XYCoord,Eelem,th,3); }
         else if(type==32){double [][]Eelem=ElasticMatrix[i];
            double []th=Thickness[i];
            Ke=Pnt6TrgMembStiff(XYCoord,Eelem,th,-3);}
         else
            return null;  if(Ke!=null)
         { for(int j=0;j<Ke.length;j++)
            { for(int k=j;k<Ke[j].length;k++)
               {  K[ongfi[j]-1][ongfi[k]-1]=K[ongfi[j]-1][ongfi[k]-
1]+Ke[j][k];
                  K[ongfi[k]-1][ongfi[j]-1]=K[ongfi[j]-1][ongfi[k]-1];}
            }
         }
         else
            return null; }
      return K;   }
```

Figure 8.2. PSGlobStrStiff module

8.3. Implementation of Reduced Global Structural Stiffness System

Support conditions of the plane stress problems have to be applied in global structural stiffness matrix to prevent singularity. The computer implementation of the boundary conditions is carried out by the ReducedGlobStrStiff module which is given in Figure 3.7. The formal parameters of the ReducedGlobStrStiff module are repeated for convenience.

Nboolean: Freedom information which contains boundary conditions

K: Global structural stiffness matrix, "K^G", which reduction is needed.

ReducedGlobStrStiff module returns $\Sigma n_{freedom} * \Sigma n_{freedom}$ reduced global structural stiffness matrix of the plane stress problems.

8.4. Implementation of External Loads

ReducedNodeForces Module in Figure 3.9 rearranges the node force matrix of the plane stress problems. The formal parameters of the ReducedNodeForces module are repeated for convenience.

Nboolean: Freedom information which contains boundary conditions

Nodevalues: Forces acting on element nodes in the same order of Nboolean.

K: Global Structural Stiffness matrix of the system

ReducedGlobStrStiff module returns $\Sigma n_{freedom} *1$ reduced node forces matrix of the plane stress problems.

8.5. Implementation of Displacement Solutions

The Reduced global structural stiffness matrix $[Kreduced]$ which contains force-displacement relationship for entire structure is used to solve all unknown displacements for any given loading scenario which were implemented in reduced node forces matrix$[freduced]$. PSSolution module shown in Figure 8.3 solves the linear system of equations of the plane stress problems.

```
public static Object [] PSSolution(double [][]allXYZcoord,int
[]elemclass,int [][]elempositions,
double [][][]ElasticMatrix, double [][]thickness, boolean
[][]Nboolean, double [][]Nodevalues)

   {double [][]K=KeCalculation.PSGlobStrStiff
(allXYZcoord,elemclass,
elempositions,ElasticMatrix,thickness);
      double [][]Kreduced=KeCalculation.ModifiedMasterStiffness
(Nboolean,K);
      double []f=new
double[Nodevalues.length*Nodevalues[0].length];
      for(int i=0,k=0;i<Nodevalues.length;i++)
      {
         for(int j=0;j<Nodevalues[i].length;j++)
         {f[k]=Nodevalues[i][j];
            k++;
      }
      double []freduced=NodeForces.ModifiedNodeForces
(Nboolean,Nodevalues,K,f);

      Jama.Matrix u=(new Jama.Matrix(Kreduced)).solve(new
Jama.Matrix(freduced,Kreduced.length));
```

Figure 8.3. PSSolution module

8.5.1. Formal Parameters of PSSolution

allXYZcoord:	All global coordinates of element nodes {(x1, y1, z1), (x2, y2, z2)…….(x_{last}, y_{last})}
elempositions:	Element end points that are assigned for the system joints. {(#1, #2, #3,#4)$_1$, (#1, #2)$_2$….. (#1, #2)$_{last}$}
ElasticMatrix:	The modulus of elasticity matrix of the plane stress elements. {(E11, E12, E13), (E21, E22, E23), (E31, E32, E33)}
elemclass:	Element class defined in Table 8.1 in the same order of "elemposition" parameter. Related numbers are indicated in Table 8.1.
Nboolean:	Freedom information which contains boundary conditions
Nodevalues:	Initial values of the forces
Thickness:	Thickness of the each element in the same order of "elemposition" parameter (th_1, $th_{2,}$….th_{last})

PSSolution module returns the unknown displacements of each node.

8.6. Implementation of Stresses of Two Dimensional Elements

8.6.1. Introduction

In the structural analysis, along with the determination of displacements, the stress prediction has also crucial importance. In the classical finite element method displacements are state variables which are continuous over a model. It is evaluated that strains and stresses are related with the following constitutive equation.

$$[\sigma] = [D]\{e\} \tag{8.2}$$

where D is the elastic matrix and {e} is the strain matrix. The stains are given by the equation (6.30).

$$\{e\} = [B]\{u_{solved}\} \tag{8.3}$$

Displacements which are solved by Jama Matrix module are used to evaluate the strain matrix. By substituting equation (8.3) into equation (8.2), the stress values of each element can be calculated. In order to evaluate the strain displacement matrix, element node locations are assigned as arguments into the Jacobi modules. Local parameter NodeCoorData contains natural coordinates of the element nodes. The implemented stress modules return the σ_{xx}, σ_{yy} and τ_{xy} stresses at element nodes.

8.6.2. Formal Parameters of Pnt4QdrMembStresses

XYcoord: Cartesian node coordinates of the quadrilateral element
{(x1, y1),(x2,y2),(x3,y3),(x4,y4)}

ElasticMatrix: The modulus of elasticity matrix of the plane stress element.
{(E11, E12, E13), (E21, E22, E23), (E31, E32, E33)}

usolved: Displacement solutions
{(u_{x1},u_{y1}) , (u_{x2},u_{y2}), (u_{x3},u_{y3}) , (u_{x4},u_{y42})}

Pnt4QdrMembStresses module in Figure 8.4., computes the σ_{xx}, σ_{yy} and τ_{xy} stresses at element nodes of the quadrilateral element sketched in Figure 6.7.

```
public static double [][] Pnt4QdrMembStresses(double
[][]XYCoord, double [][]ElasticMatrix, double []usolved)
   {  double [][] qxqytxy=MatrixOperations.zero_matrix(4,3);
      double [][]NodeCoorData={{-1,-1},{1,-1},{1,1},{-1,1}};
      double []elemdisp=new double[usolved.length];
      for(int i=0;i<usolved.length;i++)
         elemdisp[i]=usolved[i];
      for(int k=0;k<qxqytxy.length;k++)
      {double []KsiNuCoord=NodeCoorData[k];
         double
[][]result=ShapeFunction.Pnt4QdrJacobi(XYCoord,KsiNuCoord);
         double [][]BM=MatrixOperations.zero_matrix(3,8);
         for(int j=0;j<4;j++)
         { BM[0][j*2]=result[1][j];      BM[1][j*2+1]=result[2][j];
           BM[2][j*2+1]=result[1][j];  BM[2][j*2]=result[2][j]; }
qxqytxy[k]=MatrixOperations.dot_product(ElasticMatrix,
MatrixOperations.dot_product(BM,elemdisp));}
      return qxqytxy;}
```

Figure 8.4. Pnt4QdrMembStresses module

8.6.3. Formal Parameters of Pnt8QdrMembStresses

XYcoord:	Cartesian node coordinates of the quadrilateral element
	{(x1,y1),(x2,y2),(x3,y3),(x4,y4),(x5,y5),(x6,y6),(x7,y7),(x8,y8)}
ElasticMatrix:	The modulus of elasticity matrix of the plane stress element.
	{(E11, E12, E13), (E21, E22, E23), (E31, E32, E33)}
usolved :	Displacement solutions
	{(u_{x1}, u_{y1}) , (u_{x2},u_{y2})….. (u_{x8}, u_{y8})}

Pnt8QdrMembStresses module in Figure 8.5., computes the σ_{xx}, σ_{yy} and τ_{xy} stresses at element nodes of the quadrilateral element sketched in Figure 6.10.

```
public static double [][] Pnt8QdrMembStresses(
double [][]XYCoord,
double [][]ElasticMatrix,
double []usolved)

   {double [][]NodeCoorData={{-1,-1},{1,-1},{1,1},{-1,1},
                                    {0,-1},{1,0},{0,1},{-1,0}};
      double [][] qxqytxy=MatrixOperations.zero_matrix(8,3);
      for(int k=1;k<=qxqytxy.length;k++)
      {
         double []KsiNuCoord=NodeCoorData[k-1];
         double [][]result=ShapeFunction.
                              Pnt8QdrJacobi(XYCoord,KsiNuCoord);
         double [][]BM=MatrixOperations.zero_matrix(3,16);
         for(int j=0;j<8;j++)
         {
            BM[0][j*2]=result[1][j];
            BM[1][j*2+1]=result[2][j];
            BM[2][j*2+1]=result[1][j];
            BM[2][j*2]=result[2][j];
          }
         qxqytxy[k-1]=MatrixOperations.dot_product
(ElasticMatrix,MatrixOperations. dot_product(BM,usolved));
      }
      return qxqytxy;
   }
```

Figure 8.5. Pnt8QdrMembStresses module

8.6.4. Formal Parameters of Pnt9QdrMembStresses

XYcoord:	Cartesian node coordinates of the quadrilateral element {(x1,y1),(x2,y2),(x3,y3),(x4,y4),(x5,y5),(x6,y6),(x7,y7),(x8,y8), (x9,y9)}
ElasticMatrix:	The modulus of elasticity matrix of the plane stress element. {(E11, E12, E13), (E21, E22, E23), (E31, E32, E33)}
usolved :	Displacement solutions $\{(u_{x1},u_{y1}) , (u_{x2},u_{y2}) \ldots\ldots (u_{x9}, u_{y9})\}$

Pnt9QdrMembStresses module in Figure 8.6., computes the σ_{xx}, σ_{yy} and τ_{xy} stresses at element nodes of the quadrilateral element sketched in Figure 6.12.

```
public static double [][] Pnt9QdrMembStresses(double
[][]XYCoord,double [][]ElasticMatrix,double []usolved)
   {
    double [][]NodeCoorData={{-1,-1},{1,-1},{1,1},{-1,1},
                                {0,-1},{1,0},{0,1},{-1,0},{0,0}};

double [][] qxqytxy=MatrixOperations.zero_matrix(9,3);
      for(int k=1;k<=qxqytxy.length;k++)
      {
double []KsiNuCoord=NodeCoorData[k-1];
         double
[][]result=ShapeFunction.Pnt9QdrJacobi(XYCoord,KsiNuCoord);
         double [][]BM=MatrixOperations.zero_matrix(3,18);
         for(int j=0;j<9;j++)
         {
            BM[0][j*2]=result[1][j];
            BM[1][j*2+1]=result[2][j];
            BM[2][j*2+1]=result[1][j];
            BM[2][j*2]=result[2][j];
         }
         qxqytxy[k-
1]=MatrixOperations.dot_product(ElasticMatrix,MatrixOperations.
dot_product(BM,usolved));
      }
      return qxqytxy;
   }
```

Figure 8.6. Pnt9QdrMembStresses module

8.6.5. Formal Parameters of Pnt3TrgMembStresses

XYcoord: Cartesian node coordinates of the quadrilateral element
{(x1, y1),(x2,y2),(x3,y3)}

ElasticMatrix: The modulus of elasticity matrix of the plane stress element.
{(E11, E12, E13), (E21, E22, E23), (E31, E32, E33)}

usolved : Displacement solutions
{(u_{x1},u_{y1}) , (u_{x2},u_{y2}), (u_{x3},u_{y3})}

Pnt9QdrMembStresses module in Figure 8.7., computes the σ_{xx}, σ_{yy} and τ_{xy} stresses at element nodes of the quadrilateral element sketched in Figure 7.1.

```
public static double [][] Pnt3TrgMembStresses(double [][]XYCoord,
double [][]ElasticMatrix,double []usolved)
   {  double [][] qxqytxy=MatrixOperations.zero_matrix(3,3);
      double x21=XYCoord[1][0]-XYCoord[0][0];
      double x13=XYCoord[0][0]-XYCoord[2][0];
      double x32=XYCoord[2][0]-XYCoord[1][0];
      double y12=XYCoord[0][1]-XYCoord[1][1];
      double y31=XYCoord[2][1]-XYCoord[0][1];
      double y23=XYCoord[1][1]-XYCoord[2][1];
      double  A=(x21*y31-x13*y12)/2;
      double [][]BM={{y23,0,y31,0,y12,0},{0,x32,0,x13,0,x21},
{x32,y23,x13,y31,x21,y12}};
      BM=MatrixOperations.multiply_with_scalar(BM,1.0/(2*A));
      qxqytxy[0]=qxqytxy[1]=qxqytxy[2]=MatrixOperations.
dot_product(ElasticMatrix,MatrixOperations.dot_product(BM,usolved));
      return qxqytxy; }
```

Figure 8.7. Pnt3TrgMembStresses module

8.6.6. Formal Parameters of Pnt6TrgMembStresses

XYcoord: Cartesian node coordinates of the quadrilateral element
{(x1,y1),(x2,y2),(x3,y3),(x4,y4),(x5,y5),(x6,y6)}

ElasticMatrix: The modulus of elasticity matrix of the plane stress element.
{(E11, E12, E13), (E21, E22, E23), (E31, E32, E33)}

usolved : Displacement solutions
{(u_{x1},u_{y1}) , (u_{x2},u_{y2})….. (u_{x6}, u_{y6})}

Pnt6TrgMembStresses module in Figure 8.8 computes the σ_{xx}, σ_{yy} and τ_{xy} stresses at element nodes of the triangular element sketched in Figure 7.5.

```
public static double [][] Pnt6TrgMembStresses(
double [][]XYCoord,
double [][]ElasticMatrix,double []usolved)
   {
      double
[][]NodeCoorData={{1,0,0},{0,1,0},{0,0,1},{1.0/2,1.0/2,0},
                        {0,1.0/2,1.0/2},{1.0/2,0,1.0/2}};
      double [][] qxqytxy=MatrixOperations.zero_matrix(6,3);
      for(int k=1;k<=qxqytxy.length;k++)
      {
         double []KsiNuCoord=NodeCoorData[k-1];
         double
[][]result=ShapeFunction.Pnt6TrgJacobi(XYCoord,KsiNuCoord);
         double [][]BM=MatrixOperations.zero_matrix(3,12);
         for(int j=0;j<6;j++)
         {
            BM[0][j*2]=result[1][j];
            BM[1][j*2+1]=result[2][j];
            BM[2][j*2+1]=result[1][j];
            BM[2][j*2]=result[2][j];
         }
         qxqytxy[k-
1]=MatrixOperations.dot_product(ElasticMatrix,MatrixOperations.
dot_product(BM,usolved));
      }
      return qxqytxy;
   }
```

Figure 8.8. Pnt6TrgMembStresses module

8.7. Implementation of Stress Smoothing Procedure

8.7.1. Introduction

In the usual case when continuity for displacements is satisfied, raw finite element stresses are discontinuous at the inter element boundaries. Since, throughout the model continuous stress distribution is expected, discontinuity of the stress results becomes a major problem. Therefore, the theory of finite elements has been expanded to overcome

this ambiguous situation by introducing different stress smoothing procedures and stress calculation techniques.

Global smoothing and local smoothing are the two general classes of the stress smoothing procedures. If carried out over a whole finite element domain the procedure is defined as a global smoothing [12].

On the other hand, local smoothing is performed at each node or small group of nodes. One of the simplest local smoothing procedures is the averaging of stresses from neighboring elements at a particular common node. The module shown in Figure 8.9 computes the average stresses of the nodes calculated by stress modules.

8.7.2. Formal Parameters of StressAveraging

XYcoord:	All global coordinates of the system nodes $\{(x_1, y_1), (x_2, y_2).....(x_{last}, y_{last})\}$
ElasticMatrix:	The modulus of elasticity matrix of the plane stress elements. {(E11, E12, E13), (E21, E22, E23), (E31, E32, E33)}
elempositions:	Element end points that are assigned for the system joints. $\{(\#1, \#2, \#3)_1, (\#1, \#2)_2..... (\#1, \#2)_{last}\}$
elemclass:	Element class defined in Table 8.1 in the same order of "elemposition" parameter Related numbers indicated in Table 8.1.
Thickness:	Thickness of the each element in the same order of "elemposition" parameter $(th_1,th_{2,}....th_{last})$
usolved :	Displacement solutions evaluated $\{ (u_{x1},u_{y1}) , (u_{x2},u_{y2})..... (u_{xlast},u_{ylast})\}$

The local parameter "crossel" determines the total number of the elements that combined at a specific node. StressAveraging module computes the σ_{xx}, σ_{yy} and τ_{xy} stresses of the related element node by taking the average values of the corresponding interconnected element.

```
public static Object[] StressAveraging(double [][]XYZCoord, int
[]elemclass,
        int [][] elementpositions, double [][][]ElasticMatrix, double
[][]Thickness, double [][]usolved)
   { int numberofele=elementpositions.length;
      int numberofnodes=XYZCoord.length;
      int []crossel=new int[numberofnodes];
      for(int i=0;i<numberofnodes;i++)
         crossel[i]=0;

      double [][]nodeQxQyTxy=
                  MatrixOperations.zero_matrix(numberofnodes,3);
      for(int e=1;e<=numberofele;e++)
      {  int []enl=elementpositions[e-1];
         int type=elemclass[e-1];
         int k=enl.length;
         double [][]qxqytxy;
         double [][]Xycoor=new double[k][];
         for(int i=0;i<k;i++)
         {
            Xycoor[i]=new double[XYZCoord[enl[i]-1].length];
            for(int j=0;j<XYZCoord[enl[i]-1].length;j++)
               Xycoor[i][j]=XYZCoord[enl[i]-1][j];
         }
         double []dispelem=new double[k*2];
         for(int i=0;i<k;i++)
         {
            elemdisp[i*2]=usolved[enl[i]-1][0];
            elemdisp[i*2+1]=usolved[enl[i]-1][1];
         }
         if(type==40)
         {
            double [][]Eelem=ElasticMatrix[e-1];


qxqytxy=Pnt4QdrMembStresses(Xycoor,Eelem,elemdisp);
         }
         else if(type==42)
         {
            double [][]Eelem=ElasticMatrix[e-1];


qxqytxy=Pnt8QdrMembStresses(Xycoor,Eelem,elemdisp);
         }
         else if(type==44)
         {
            double [][]Eelem=ElasticMatrix[e-1];
```

```
qxqytxy=Pnt9QdrMembStresses(Xycoor,Eelem,elemdisp);
          }
          else if(type==30)
          {
             double [][]Eelem=ElasticMatrix[e-1];

qxqytxy=Pnt3TrgMembStresses(Xycoor,Eelem,elemdisp);
          }
          else if(type==31)
          {
             double [][]Eelem=ElasticMatrix[e-1];

qxqytxy=Pnt6TrgMembStresses(Xycoor,Eelem,elemdisp);
          }
          else if(type==32)
          {
             double [][]Eelem=ElasticMatrix[e-1];

qxqytxy=Pnt6TrgMembStresses(Xycoor,Eelem,elemdisp);
          }
          else
             continue;
          for(int i=1;i<=k;i++)
          {
             int n=enl[i-1];
             crossel[n-1]++;
             for(int j=0;j<nodeQxQyTxy[n-1].length;j++)
             {
                nodeQxQyTxy[n-1][j]+=qxqytxy[i-1][j];
             }
          }
       }
       for(int i=1;i<numberofnodes;i++)
       {
          double k=crossel[i-1];
          if(k>1)
          {
             for(int j=0;j<nodeQxQyTxy[i-1].length;j++)
                nodeQxQyTxy[i-1][j]/=k;
          }
       }
       Object []result=new Object[2];
       result[0]=crossel;
       result[1]=nodeQxQyTxy;
       return result;
    }
```

Figure 8.9. StressAveraging module

8.8. Implementation of Results

Java object called Results which is shown in Figure 8.10 collects the solutions of plane stress problems. Results of the system are gathered with the four local parameters and transmitted to the Output Handler.

Local parameter “usolved” collects the displacement solution matrix which is evaluated by Jama.Matrix module. Local parameter “NForces” collects forces of the each node. Finally, local parameter “nodeQxQyTxy” collects stress components of the two dimensional elements by calling “StressAveraging” module.

```
double []my_u=new double[Kreduced.length];
      for(int i=0;i<my_u.length;i++)
         my_u[i]=(u.getArray())[i][0];
      for(int i=0;i<K.length;i++)
      {
         f[i]=MatrixOperations.dot_product(K[i],my_u);
      }
      double [][]Nforces=MatrixOperations.mflatting(2,f);
      double [][]usolved=MatrixOperations.mflatting(2,my_u);
      Object []nodeQxQyTxy=StressSolutions.StressAveraging

(allXYZcoord,elemclass,elempositions,ElasticMatrix,thickness,usolved);
      Object []results=new Object[3];
      results[0]=usolved;
      results[1]=Nforces;
      results[2]=(double [][])(nodeQxQyTxy[1]);
      return results;    }
```

Figure 8.10. Results object for plane stress problems

8.9. Flow Chart of the Plane Stress Program

Schematic representation of the algorithm of the plane stress program is given in Figure 8.11.

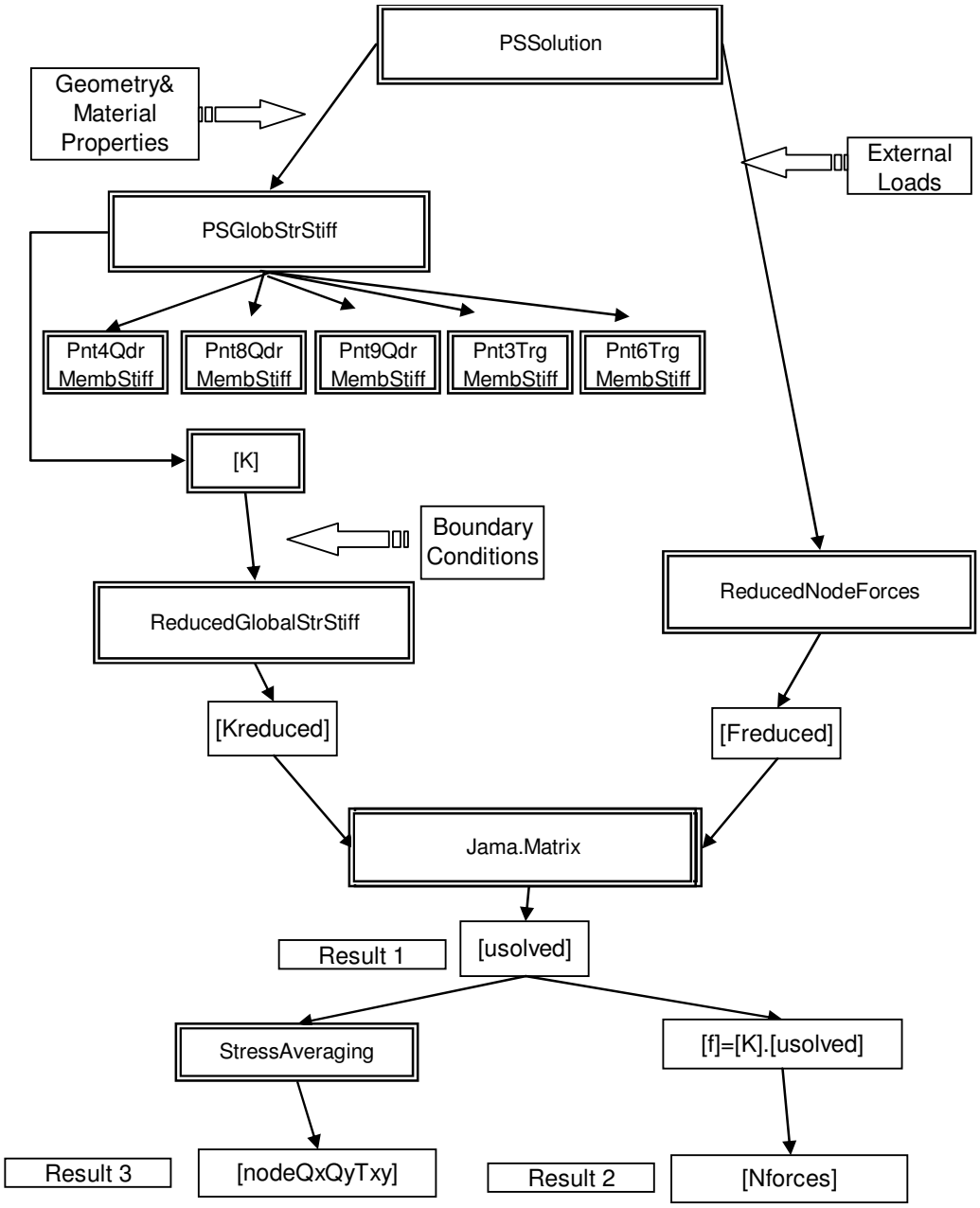


Figure 8.11. Flowchart of the plane stress program

9. PLANE STRESS CASE OF A CANTILEVER BEAM

9.1. Introduction

The purpose of this section is to compare Java plane stress program with the LUSAS finite element software. The plane stress elements Pnt3Trg, Pnt6Trg, Pnt4Quad and Pnt8Quad will be tested with the same plane stress elements introduced in LUSAS library.

The cantilever thin plate sketched in Figure 9.1 is subjected to vertical P force which is distributed parabolicly along the section I-I.

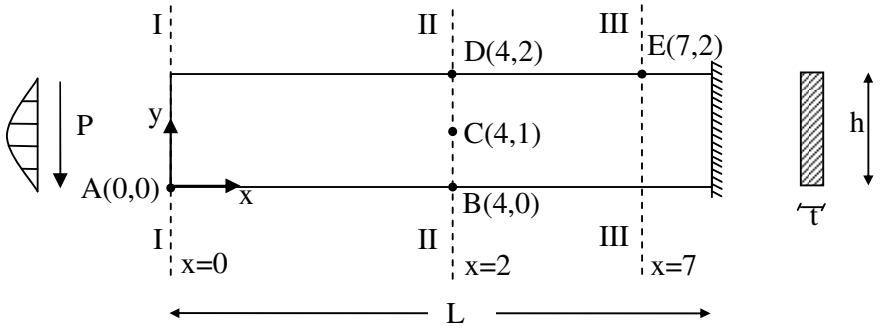


Figure 9.1. Cantilever beam plane stress problem

The material and geometric properties of the problem are given in the Table 9.1.

Table 9.1. Properties of the plane truss problem

Force (P)	40 kN
Length (L)	8 m
Thickness (t)	0.2m
Height(h)	2m
Elastic Modulus (E)	$3.\ 10^7$ kN / m^2
Poisson's ratio (ν)	0.3

9.2. Determination of the Exact Solutions

According to the theory of the elasticity, the exact values of stresses for points A, B, C, D and E are evaluated by the following formulas [18] ;

$$\sigma_x = -\frac{Pxy}{I_{zz}} \ , \ \sigma_y = 0 \ , \ \tau_{xy} = -\frac{P}{2I_{zz}}(\frac{h^2}{4} - y^2) \tag{9.1}$$

where I_{zz} is the moment of inertia and x,y are the local coordinates illustrated in Figure 9.2.

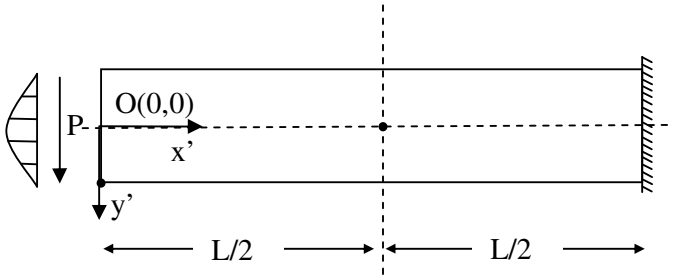


Figure 9.2. Local coordinates for exact solution formulas

The exact values of the vertical and horizontal displacements along the cantilever beam can be evaluated by equation 9.2.

$$\begin{aligned} u_x &= -\frac{Pyx^2}{2EI_{ZZ}} - \frac{\nu Py^3}{6EI_{ZZ}} + \frac{Py^3}{6GI_{ZZ}} + y(\frac{PL^2}{2EI_{ZZ}} - \frac{Ph^2}{8GI_{ZZ}}) \\ u_y &= \frac{\nu Pxy^2}{2EI_{ZZ}} + \frac{Px^3}{6EI_{ZZ}} - \frac{PL^2x}{2EI_{ZZ}} + \frac{PL^3}{3EI_{ZZ}} + \frac{Pc^2}{2IG}(L-x) \end{aligned} \tag{9.2}$$

where ν is the poissons ratio and G is the shear modulus.

The numerical solutions of stresses and deflections at points A, B, C, D and E are given in the Table 9.2.

Table 9.2. The exact values of stresses and deflections

Points	x (m)	y (m)	σ_x (kN/m)	σ_y (kN/m)	τ_{xy} (kN/m)	u_x (10^{-4}m)	u_y (10^{-3}m)
A	0	0	0	0	0	3.108	1.811
B	4	0	-1200	0	0	2.308	0.591
C	4	1	0	0	-150	0.000	0.585
D	4	2	1200	0	0	-2.308	0.591
E	7	2	2100	0	0	-0.658	0.062

9.3. Data Entry

In the context of data processing, data are defined as numbers that represent measurements from observable phenomena. In the case of the plane stress problem introduced in Figure 9.1, mathematical model data of the cantilever beam is prepared by the template file given in the Appendix B. The plane stress template file for the Java program is named "Plane_stress_input.xls" and can be found in the Templates folder of the Appendix B.

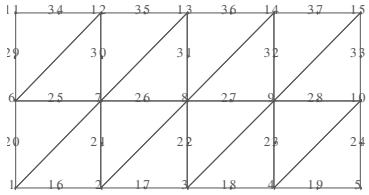
In the first step, the cantilever beam is subdivided into a coarse mesh of 2 by 4. The coordinates of the nodes and corresponding node constrains are determined by Figure 9.1. The values of the distributed loads along the section I-I are obtained by the following equation [19].

$$P = \int_a^b \frac{2P}{h^3}\left(3y^2 - 0.75h^2\right)dy \tag{9.3}$$

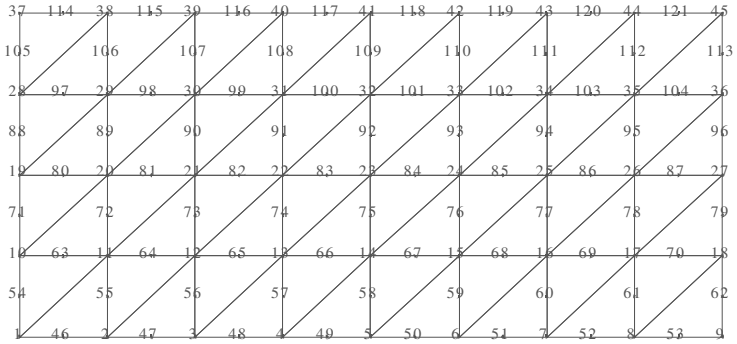
The modulus of elasticity matrix is obtained by the equation (9.4).

$$\left[E_{mat}\right] = \frac{E}{1-\upsilon^2}\begin{bmatrix} 1 & -\nu & 0 \\ -\nu & 1 & 0 \\ 0 & 0 & \dfrac{1-\nu}{2} \end{bmatrix} \tag{9.4}$$

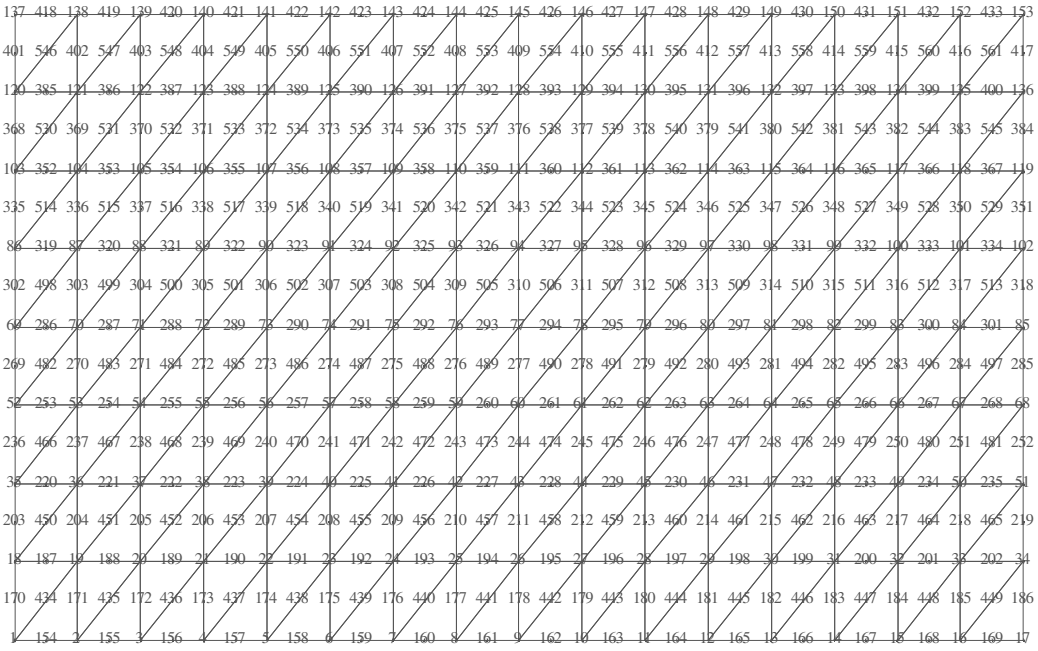
Element types are determined from the Table 8.1. Finally, the node numbers of the elements are determined by the mesh figures illustrated in Figure 9.3 and Figure 9.4.



2/4 Mesh Size

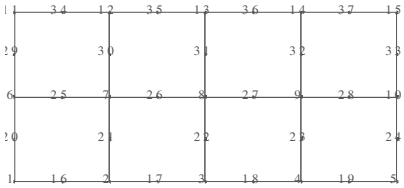


4/8 Mesh Size

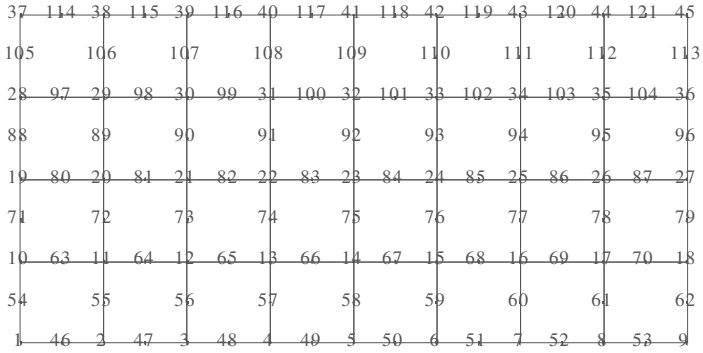


8/16 Mesh Size

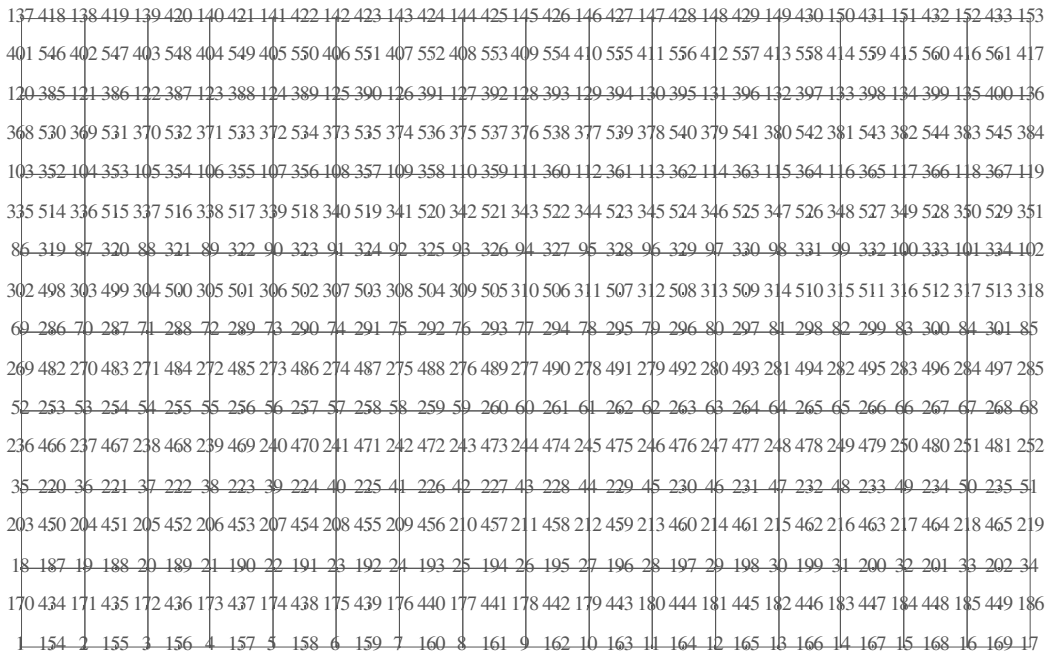
Figure 9.3. Triangular mesh node points of cantilever beam case



2/4 Mesh Size



4/8 Mesh Size



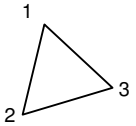
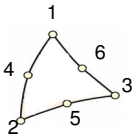
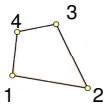
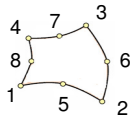
8/16 Mesh Size

Figure 9.4. Quadrilateral mesh node points of cantilever beam case

All data for the 2x4 mesh is entered into the related template files given in Appendix B. Four different element types are used for 2x4 meshes; therefore 4 different input files are generated.

In the second step, the mesh sizes are increased to 4x8. Finally in the third step the problem is subdivided into 8x16 meshes. Four different element types are used again for each mesh cases; therefore 8 different input files are generated. The name of the input files and their mesh sizes are shown in Table 9.3.

Table 9.3. Input file names given in Appendix B for cantilever beam case

Name	**Mesh Element**	**Input File Name**		
		2X4 Mesh	4X8 Mesh	8x16 Mesh
Pnt3Trg		plane_stress_input 2x4trig3.xls	plane_stress_input 4x8trig3.xls	plane_stress_input 8x16trig3.xls
Pnt6Trg		plane_stress_input 2x4trig6.xls	plane_stress_input 4x8trig6.xls	plane_stress_input 8x16trig6.xls
Pnt4Quad		plane_stress_input 2x4quad4.xls	plane_stress_input 4x8quad4.xls	plane_stress_input 8x16quad4.xls
Pnt8Quad		plane_stress_input 2x4quad8.xls	plane_stress_input 4x8quad8.xls	plane_stress_input 8x16quad8.xls

9.4. Comparison of the Results with LUSAS Software

Lusas is one of the leading industrial software for undertaking finite element analysis to ensure that components are capable of withstanding a pre-defined loading. Lusas is used throughout the world by engineers in the construction industry for all types of civil and structural design. The results of the cantilever plane stress problem with Java program introduced previously will be compared with the Lusas software solutions [20].

Horizontal and vertical displacements of the points A and D are determined and the comparison tables are shown in Table 9.4 to Table 9.11. In order to compare the performance of the Java program, accuracies of the results by the triangular and quadrilateral elements are plotted in two different figures and results are illustrated in Figure 9.5 to Figure 9.12.

The normal stress components of the critical points D and E are determined and the comparison tables are shown in Table 9.12 to Table 9.15. In order to compare the performance of the Java program with Lusas software, accuracies of the results by the triangular and quadrilateral elements are plotted in two different figures and results are illustrated in Figure 9.13 to Figure 9.16.

Finally, the shear stress component of the point C is determined and matched with the Lusas solution through Table 9.17 and Tables 9.18. Accuracy comparison is illustrated in Figure 9.17 to Figure 9.18.

Table 9.4. Comparison of vertical deflection at point A

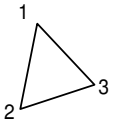
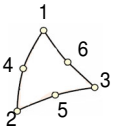
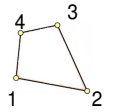
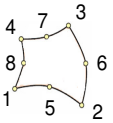
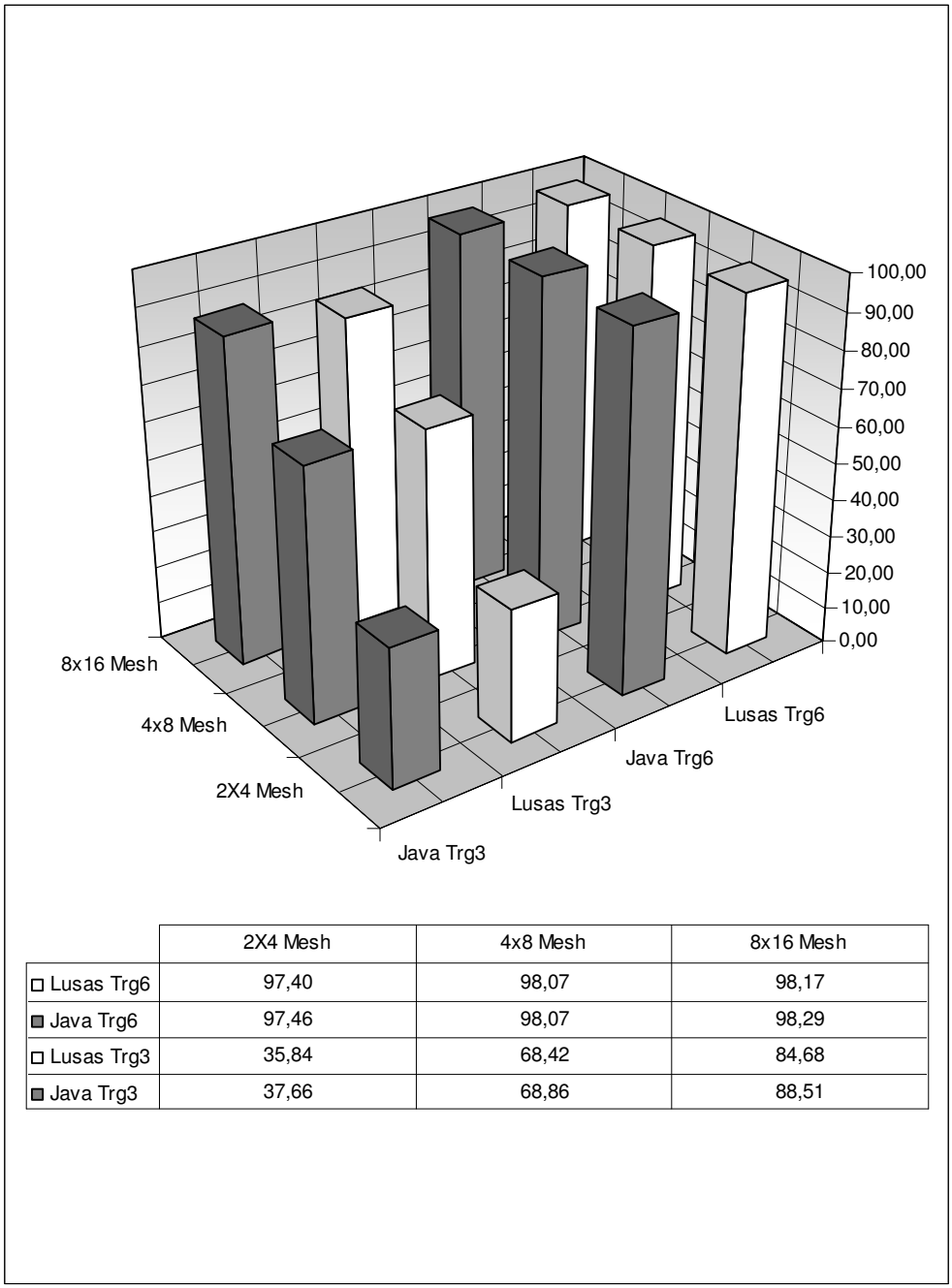
Vertical Deflection(δ_y) at x=0,y=0 (m)(10^{-3})							
Name	**Geometry**	**Solution**					
		2X4 Mesh		4x8 Mesh		8x16 Mesh	
		Lusas 13	Java	Lusas 13	Java	Lusas 13	Java
Pnt3Trg		0.649	0.682	1.239	1.247	1.5336	1.603
Pnt6Trg		1.764	1.765	1.776	1.776	1.7778	1.780
Pnt4Quad		1.167	1.248	1.600	1.604	1.6996	1.777
Pnt8Quad		1.775	1.763	1.777	1.777	1.7782	1.779

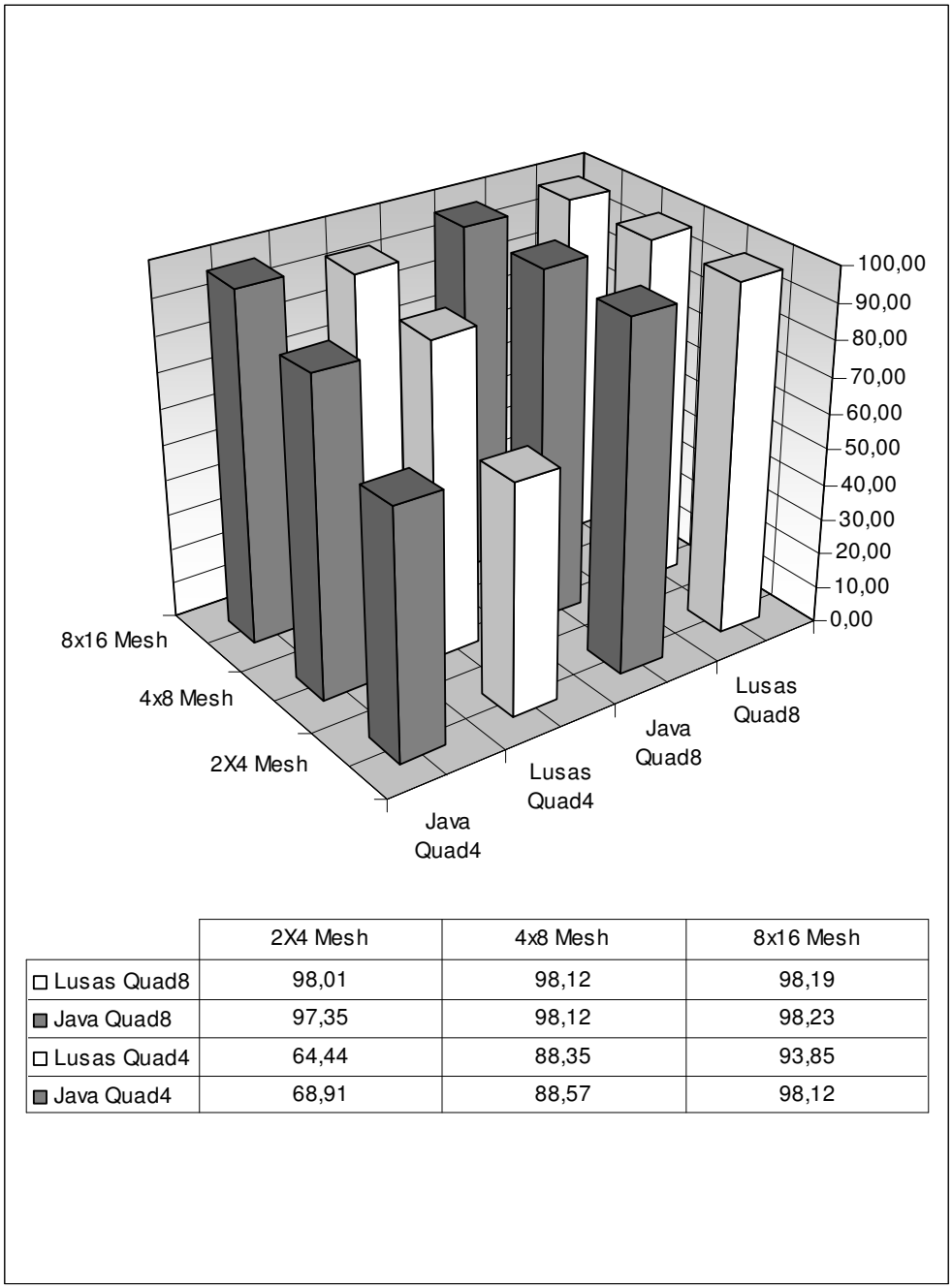
Table 9.5. Comparison of accuracies of vertical deflection at point A

δy Accuracy at x=0,y=0 (%)							
Name	**Exact Solution**	**Accuracy(%)**					
		2X4 Mesh		4x8 Mesh		8x16 Mesh	
		Lusas	Java	Lusas	Java	Lusas	Java
Pnt3Trg	1.811	35.84	37.66	68.42	68.86	84.68	88.51
Pnt6Trg	1.811	97.40	97.46	98.07	98.07	98.17	98.29
Pnt4Quad	1.811	64.44	68.91	88.35	88.57	93.85	98.12
Pnt8Quad	1.811	98.01	97.35	98.12	98.12	98.19	98.23



	2X4 Mesh	4x8 Mesh	8x16 Mesh
Lusas Trg6	97,40	98,07	98,17
Java Trg6	97,46	98,07	98,29
Lusas Trg3	35,84	68,42	84,68
Java Trg3	37,66	68,86	88,51

Figure 9.5. δ_y Accuracy of triangular elements at point A



	2X4 Mesh	4x8 Mesh	8x16 Mesh
Lusas Quad8	98,01	98,12	98,19
Java Quad8	97,35	98,12	98,23
Lusas Quad4	64,44	88,35	93,85
Java Quad4	68,91	88,57	98,12

Figure 9.6. δ_y Accuracy of quadrilateral elements at point A

Table 9.6. Comparison of horizontal deflection at point A

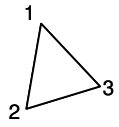
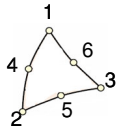
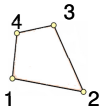
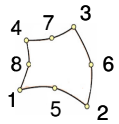
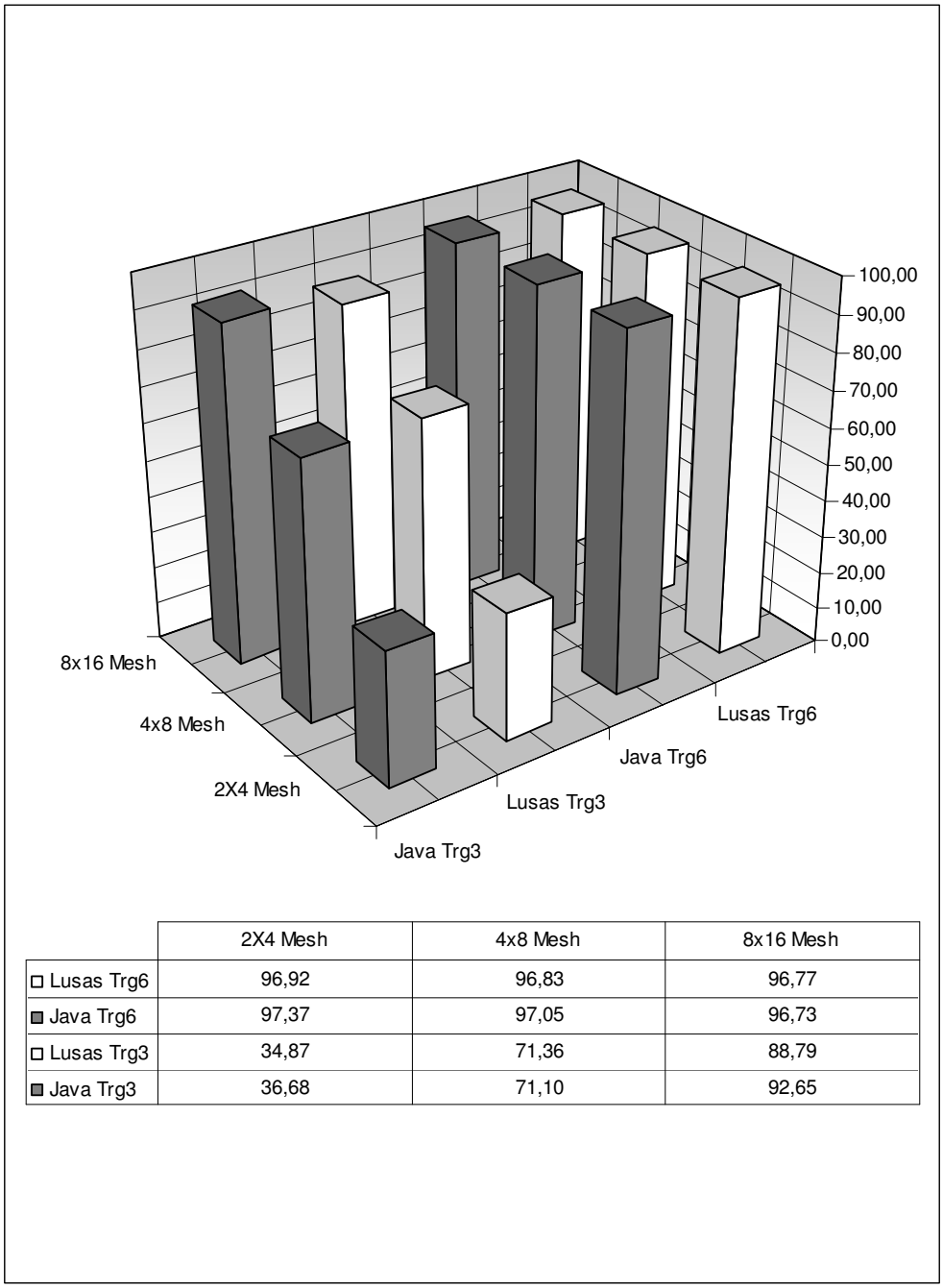
Horizontal Deflection(δ_x) at x=0,y=0 (m)(10^{-4})							
Name	Geometry	Solution					
		2X4 Mesh		4x8 Mesh		8x16 Mesh	
		Lusas	Java	Lusas	Java	Lusas	Java
Pnt3Trg		1.084	1.140	2.218	2.210	2.7598	2.880
Pnt6Trg		3.204	3.190	3.207	3.200	3.2088	3.210
Pnt4Quad		2.070	2.252	2.891	2.892	3.0686	3.119
Pnt8Quad		3.205	3.187	3.208	3.200	3.2092	3.200

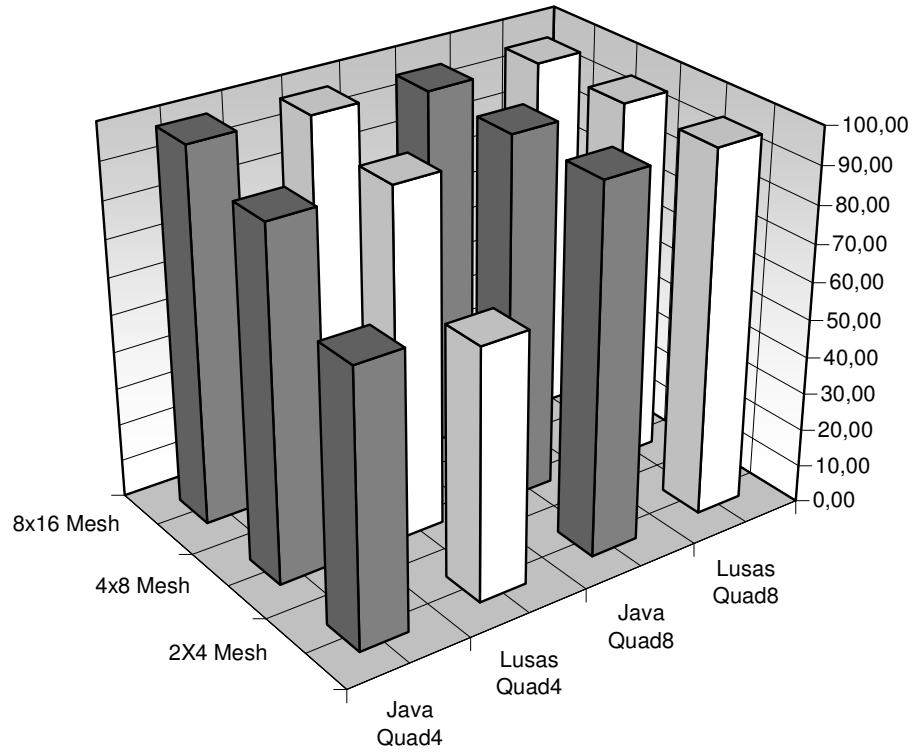
Table 9.7. Comparison of accuracies of horizontal deflection at point A

(δ_x) Accuracy at x=0,y=0 (%)							
Name	Exact Solution	Accuracy(%)					
		2X4 Mesh		4x8 Mesh		8x16 Mesh	
		Lusas	Java	Lusas	Java	Lusas	Java
Pnt3Trg	3.1083	34.87	36.68	71.36	71.10	88.79	92.65
Pnt6Trg	3.1083	96.92	97.37	96.83	97.05	96.77	96.73
Pnt4Quad	3.1083	66.60	72.45	93.01	93.04	98.72	99.66
Pnt8Quad	3.1083	96.89	97.47	96.79	97.05	96.75	97.05



	2X4 Mesh	4x8 Mesh	8x16 Mesh
Lusas Trg6	96,92	96,83	96,77
Java Trg6	97,37	97,05	96,73
Lusas Trg3	34,87	71,36	88,79
Java Trg3	36,68	71,10	92,65

Figure 9.7. δ_x Accuracy of triangular elements at point A



	2X4 Mesh	4x8 Mesh	8x16 Mesh
Lusas Quad8	96,89	96,79	96,75
Java Quad8	97,47	97,05	97,05
Lusas Quad4	66,60	93,01	98,72
Java Quad4	72,45	93,04	99,66

Figure 9.8. δ_x Accuracy of quadrilateral elements at point A

Table 9.8. Comparison of vertical deflection at point D

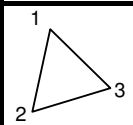
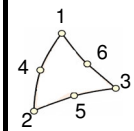
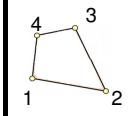
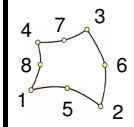
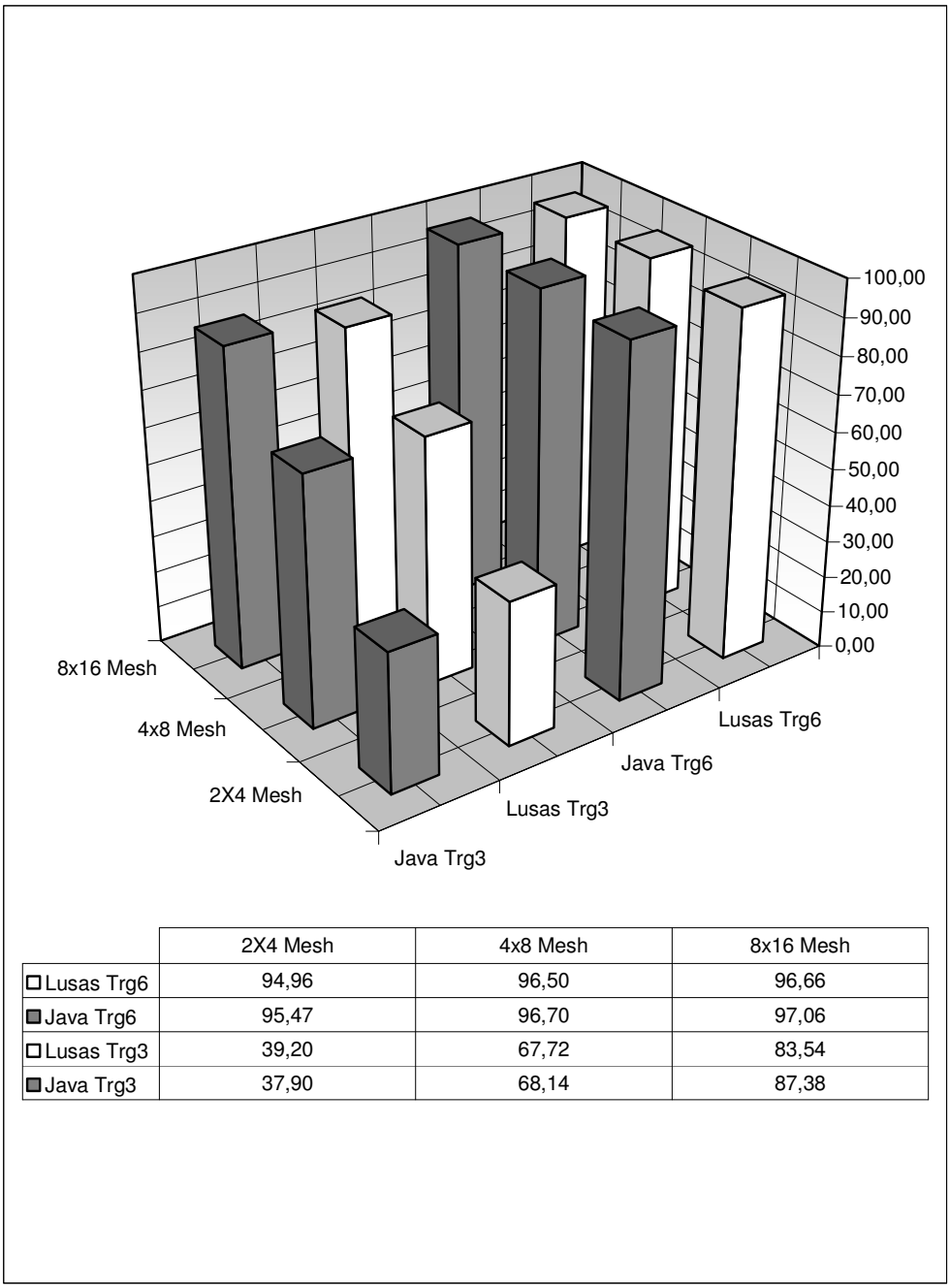
Vertical Deflection(δ_y) at x=4,y=0 (m)(10^{-4})							
Name	**Geometry**	**Solution**					
		2X4 Mesh		4x8 Mesh		8x16 Mesh	
		Lusas	Java	Lusas	Java	Lusas	Java
Pnt3Trg		2.318	2.241	4.004	4.029	4.940	5.167
Pnt6Trg		5.615	5.645	5.706	5.718	5.716	5.739
Pnt4Quad		3.919	3.995	5.133	5.154	5.461	5.577
Pnt8Quad		5.698	5.650	5.714	5.722	5.719	5.740

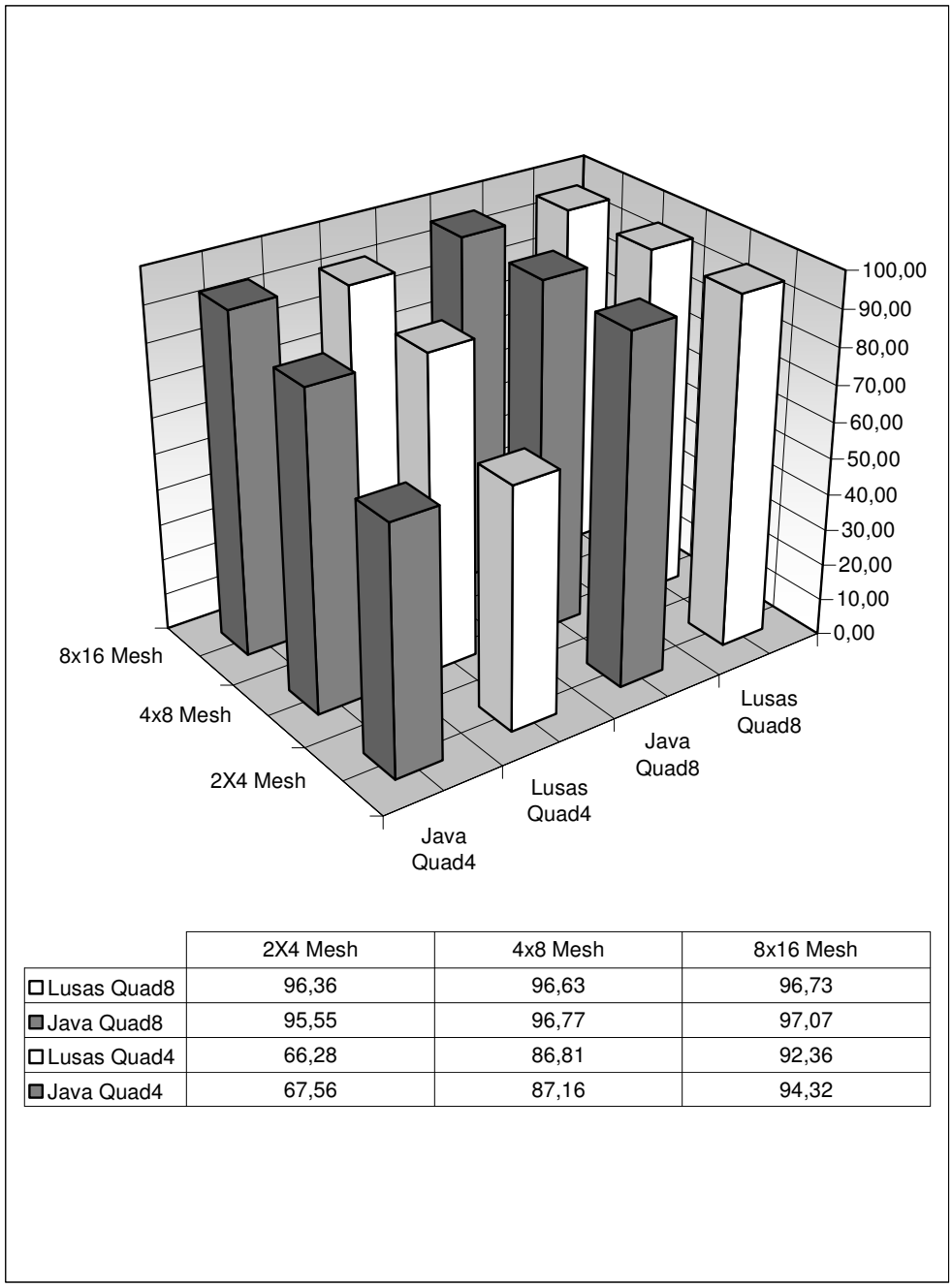
Table 9.9. Comparison of accuracies of vertical deflection at point D

Accuracy (δ_y) at x=4,y=0 (%)							
Name	**Exact Solution**	**Accuracy(%)**					
		2X4 Mesh		4x8 Mesh		8x16 Mesh	
		Lusas	Java	Lusas	Java	Lusas	Java
Pnt3Trg	5.913	39.20	37.90	67.72	68.14	83.54	87.38
Pnt6Trg	5.913	94.96	95.47	96.50	96.70	96.66	97.06
Pnt4Quad	5.913	66.28	67.56	86.81	87.16	92.36	94.32
Pnt8Quad	5.913	96.36	95.55	96.63	96.77	96.73	97.07



	2X4 Mesh	4x8 Mesh	8x16 Mesh
Lusas Trg6	94,96	96,50	96,66
Java Trg6	95,47	96,70	97,06
Lusas Trg3	39,20	67,72	83,54
Java Trg3	37,90	68,14	87,38

Figure 9.9. δ_y Accuracy of triangular elements at point D



	2X4 Mesh	4x8 Mesh	8x16 Mesh
Lusas Quad8	96,36	96,63	96,73
Java Quad8	95,55	96,77	97,07
Lusas Quad4	66,28	86,81	92,36
Java Quad4	67,56	87,16	94,32

Figure 9.10. δ_y Accuracy of quadrilateral elements at point D

Table 9.10. Comparison of horizontal deflection at point D

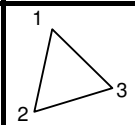
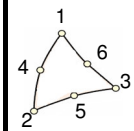
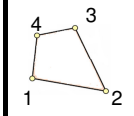
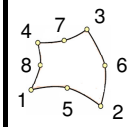
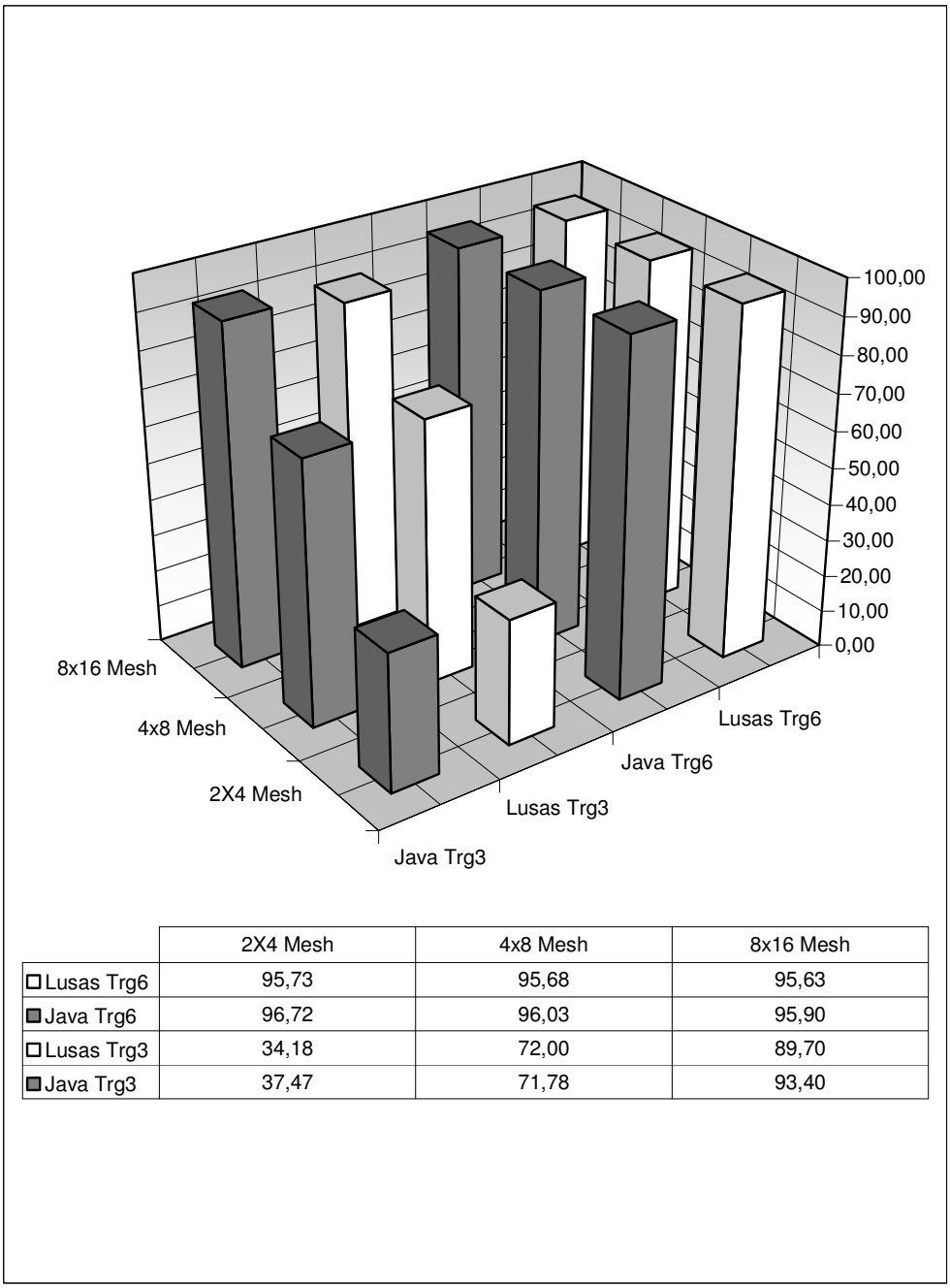
Horizontal Deflection(δ_x) at x=4,y=0 (m)(10^{-4})							
Name	**Geometry**	**Solution**					
		2X4 Mesh		4x8 Mesh		8x16 Mesh	
		Lusas	Java	Lusas	Java	Lusas	Java
Pnt3Trg		0.789	0.865	1.662	1.657	2.0706	2.156
Pnt6Trg		2.407	2.384	2.408	2.400	2.4092	2.403
Pnt4Quad		1.511	1.686	2.168	2.166	2.3012	2.338
Pnt8Quad		2.406	2.390	2.408	2.400	2.4092	2.400

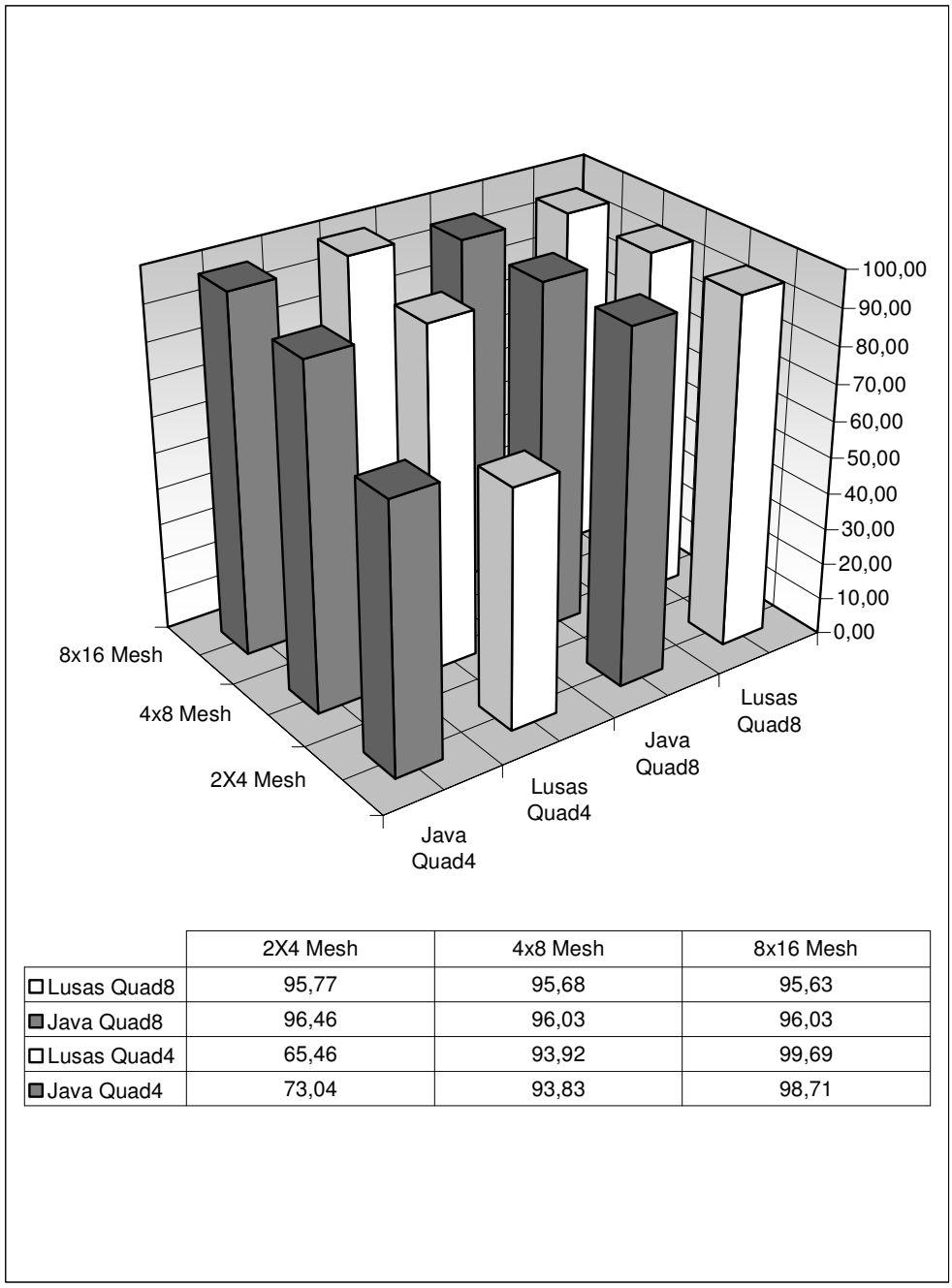
Table 9.11. Comparison of accuracies of horizontal deflection at point D

Accuracy at x=4,y=0 (%)							
Name	**Exact Solution**	**Accuracy(%)**					
		2X4 Mesh		4x8 Mesh		8x16 Mesh	
		Lusas	Java	Lusas	Java	Lusas	Java
Pnt3Trg	2.3083	34.18	37.47	72.00	71.78	89.70	93.40
Pnt6Trg	2.3083	95.73	96.72	95.68	96.03	95.63	95.90
Pnt4Quad	2.3083	65.46	73.04	93.92	93.83	99.69	98.71
Pnt8Quad	2.3083	95.77	96.46	95.68	96.03	95.63	96.03



	2X4 Mesh	4x8 Mesh	8x16 Mesh
Lusas Trg6	95,73	95,68	95,63
Java Trg6	96,72	96,03	95,90
Lusas Trg3	34,18	72,00	89,70
Java Trg3	37,47	71,78	93,40

Figure 9.11. δ_x Accuracy of triangular elements at point D



	2X4 Mesh	4x8 Mesh	8x16 Mesh
Lusas Quad8	95,77	95,68	95,63
Java Quad8	96,46	96,03	96,03
Lusas Quad4	65,46	93,92	99,69
Java Quad4	73,04	93,83	98,71

Figure 9.12. δ_x Accuracy of quadrilateral elements at point D

Table 9.12. Comparison of shear stress at point C

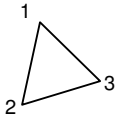
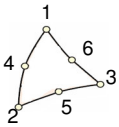
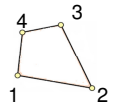
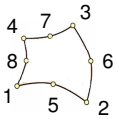
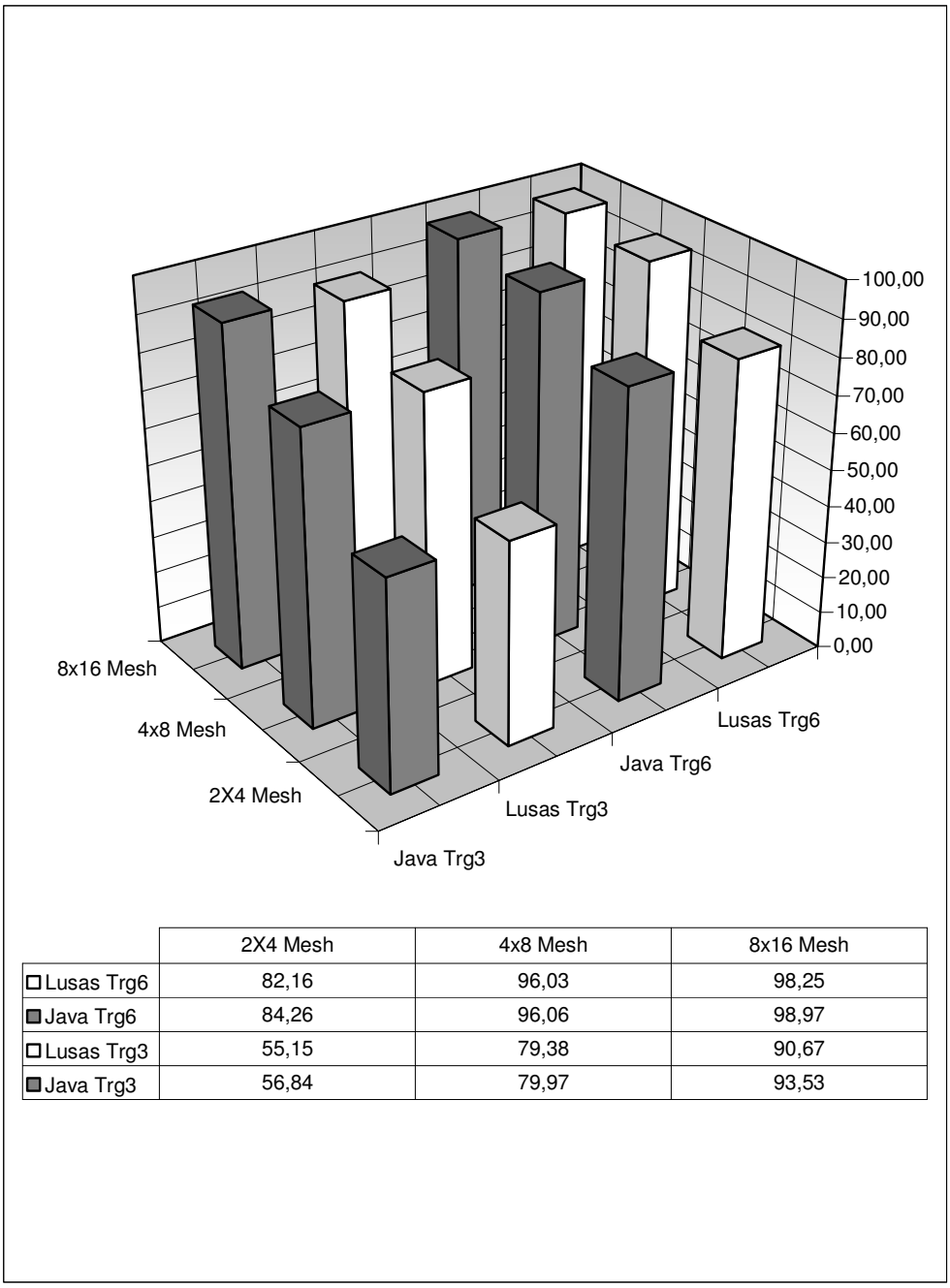
Shear Stress(τ_{xy}) at x=4,y=1 (kN/m^2)							
Name	**Geometry**	**Solution**					
		2X4 Mesh		4x8 Mesh		8x16 Mesh	
		Lusas	Java	Lusas	Java	Lusas	Java
Pnt3Trg		82.729	85.260	119.070	119.960	135.998	140.290
Pnt6Trg		176.756	173.610	155.960	155.910	152.619	151.550
Pnt4Quad		-31.134	25.146	106.978	109.713	131.021	139.202
Pnt8Quad		171.495	207.980	155.794	165.180	152.552	153.780

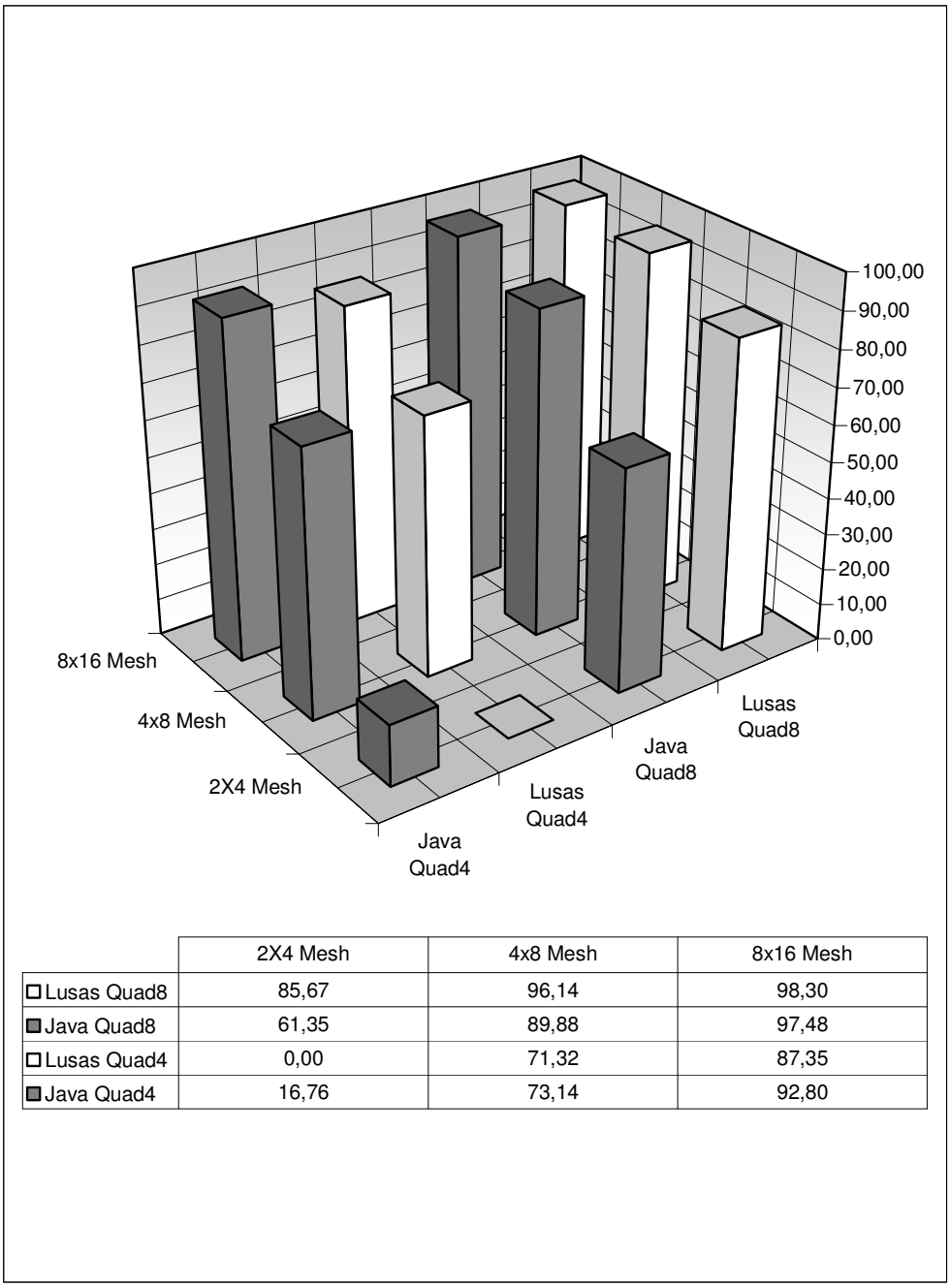
Table 9.13. Comparison of accuracies of shear stress at point C

Shear Stress Accuracy (τ_{xy}) at x=4,y=1 (%)							
Name	**Exact Solution**	**Accuracy(%)**					
		2X4 Mesh		4x8 Mesh		8x16 Mesh	
		Lusas	Java	Lusas	Java	Lusas	Java
Pnt3Trg	150	55.15	56.84	79.38	79.97	90.67	93.53
Pnt6Trg	150	82.16	84.26	96.03	96.06	98.25	98.97
Pnt4Quad	150	0	16.76	71.32	73.14	87.35	92.80
Pnt8Quad	150	85.67	61.35	96.14	89.88	98.30	97.48



	2X4 Mesh	4x8 Mesh	8x16 Mesh
Lusas Trg6	82,16	96,03	98,25
Java Trg6	84,26	96,06	98,97
Lusas Trg3	55,15	79,38	90,67
Java Trg3	56,84	79,97	93,53

Figure 9.13. τ_{xy} Accuracy of triangular elements at point C



	2X4 Mesh	4x8 Mesh	8x16 Mesh
Lusas Quad8	85,67	96,14	98,30
Java Quad8	61,35	89,88	97,48
Lusas Quad4	0,00	71,32	87,35
Java Quad4	16,76	73,14	92,80

Figure 9.14. τ_{xy} Accuracy of quadrilateral elements at point C

Table 9.14. Comparison of normal stress at point D

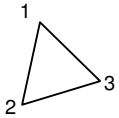
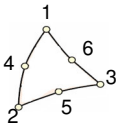
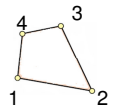
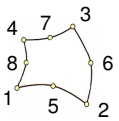
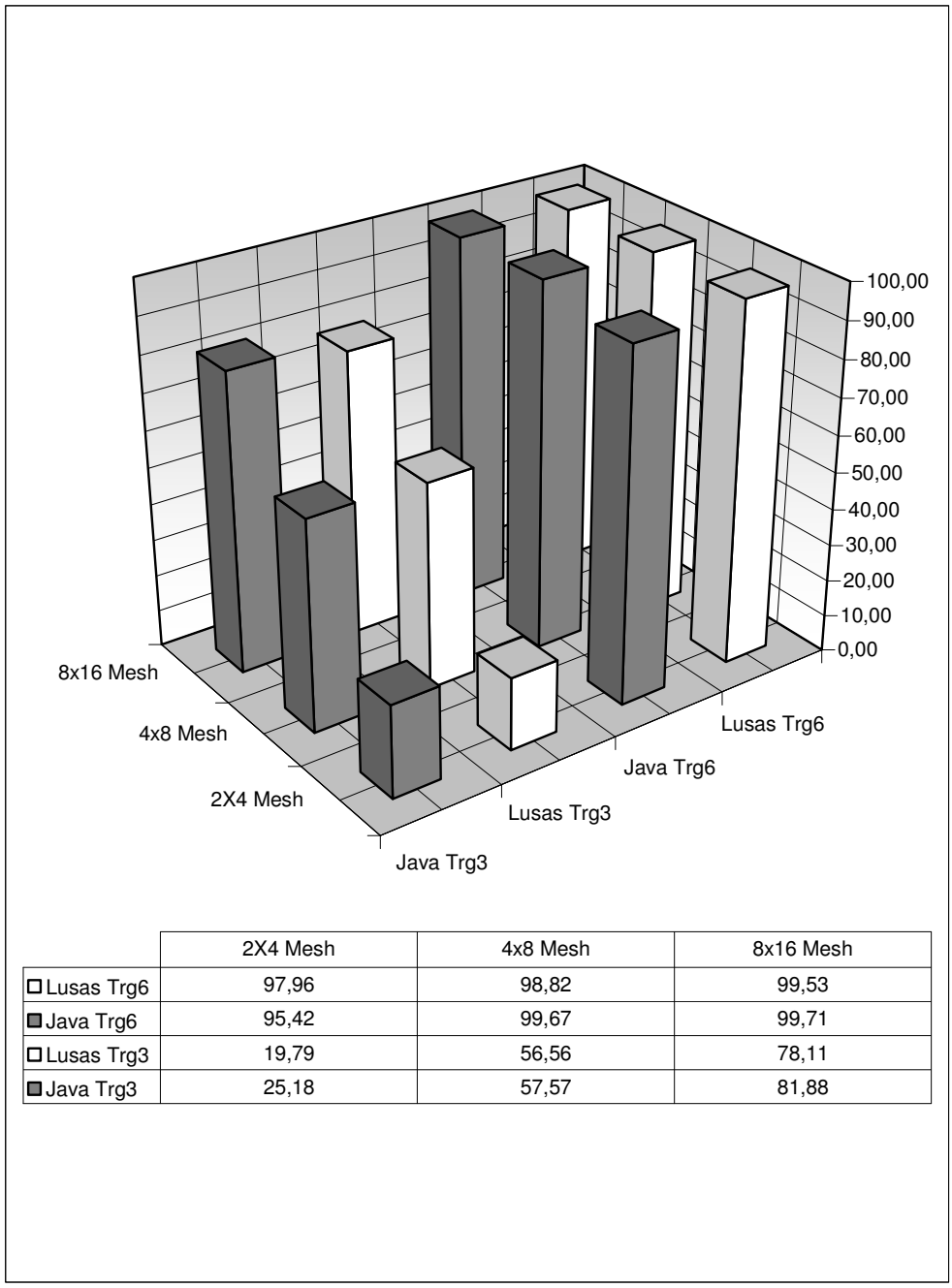
Normal Stress(σ_x) at x=4,y=2 (kN/m^2)							
Name	**Geometry**	**Solution**					
		2X4 Mesh		4x8 Mesh		8x16 Mesh	
		Lusas	Java	Lusas	Java	Lusas	Java
Pnt3Trg		237.51	302.174	678.71	690.814	937.334	982.61
Pnt6Trg		1175.49	1145.07	1185.85	1195.98	1194.34	1196.50
Pnt4Quad		757.53	901.81	1095.46	1115.57	1155.18	1206.61
Pnt8Quad		1200.95	1205.55	1200.01	1199.97	1200	1200.00

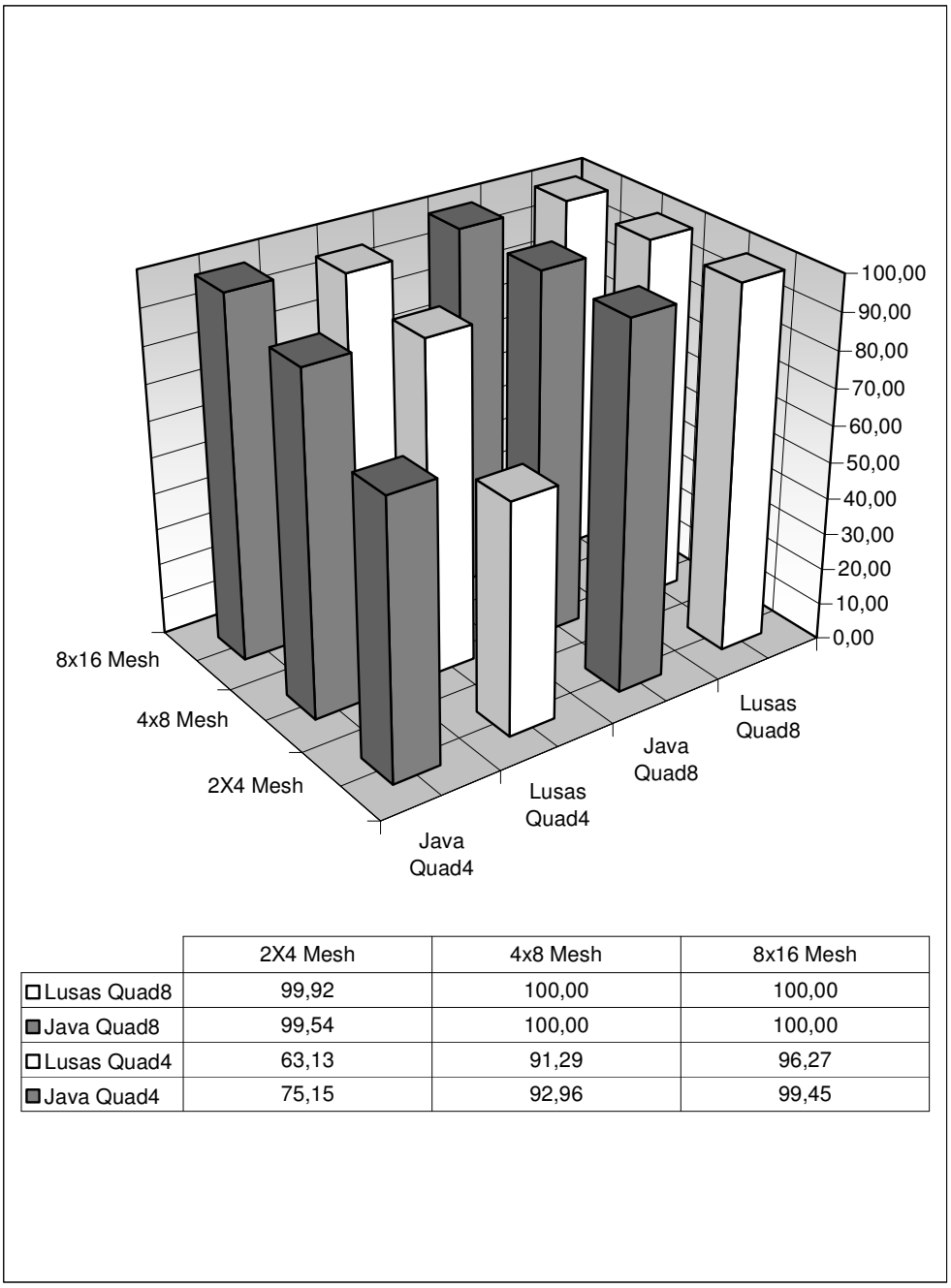
Table 9.15. Comparison of accuracies of normal stress at point D

Normal Stress(σ_x) Accuracy at x=4,y=2 (%)							
Name	**Exact Solution**	**Accuracy(%)**					
		2X4 Mesh		4x8 Mesh		8x16 Mesh	
		Lusas	Java	Lusas	Java	Lusas	Java
Pnt3Trg	1200	19.79	25.18	56.56	57.57	78.11	81.88
Pnt6Trg	1200	97.96	95.42	98.82	99.67	99.53	99.71
Pnt4Quad	1200	63.13	75.15	91.29	92.96	96.27	99.45
Pnt8Quad	1200	99.92	99.54	100.00	100.00	100.00	100.00



	2X4 Mesh	4x8 Mesh	8x16 Mesh
Lusas Trg6	97,96	98,82	99,53
Java Trg6	95,42	99,67	99,71
Lusas Trg3	19,79	56,56	78,11
Java Trg3	25,18	57,57	81,88

Figure 9.15. σ_x Accuracy of triangular elements at point D



	2X4 Mesh	4x8 Mesh	8x16 Mesh
Lusas Quad8	99,92	100,00	100,00
Java Quad8	99,54	100,00	100,00
Lusas Quad4	63,13	91,29	96,27
Java Quad4	75,15	92,96	99,45

Figure 9.16. σ_x Accuracy of quadrilateral elements at point D

Table 9.16. Comparison of normal stress at point E

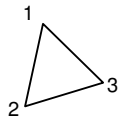
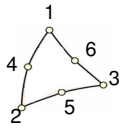
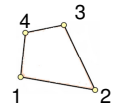
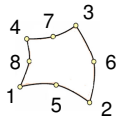
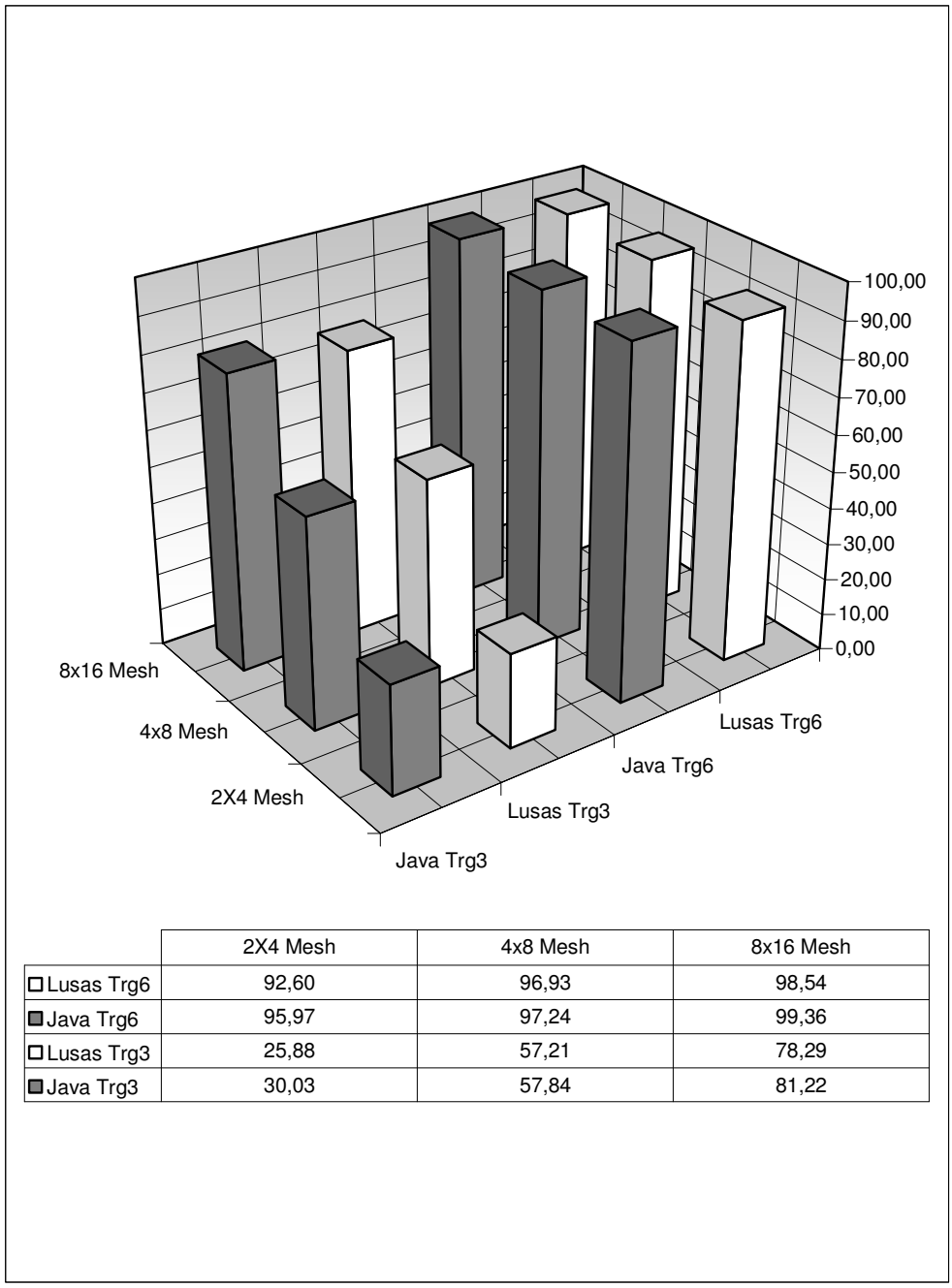
Normal Stress(σ_x) at x=7,y=2 (kN/m2)							
Name	**Geometry**	**Solution**					
		2X4 Mesh		4x8 Mesh		8x16 Mesh	
		Lusas	Java	Lusas	Java	Lusas	Java
Pnt3Trg		543.56	630.63	1201.50	1214.64	1644.18	1705.64
Pnt6Trg		1944.6	2015.43	2035.50	2042.11	2069.4	2086.47
Pnt4Quad		1445.75	1453.5	1903.49	1903.76	2010.6	2046.28
Pnt8Quad		2006.52	2118.56	2062.18	2085.02	2080.67	2086.06

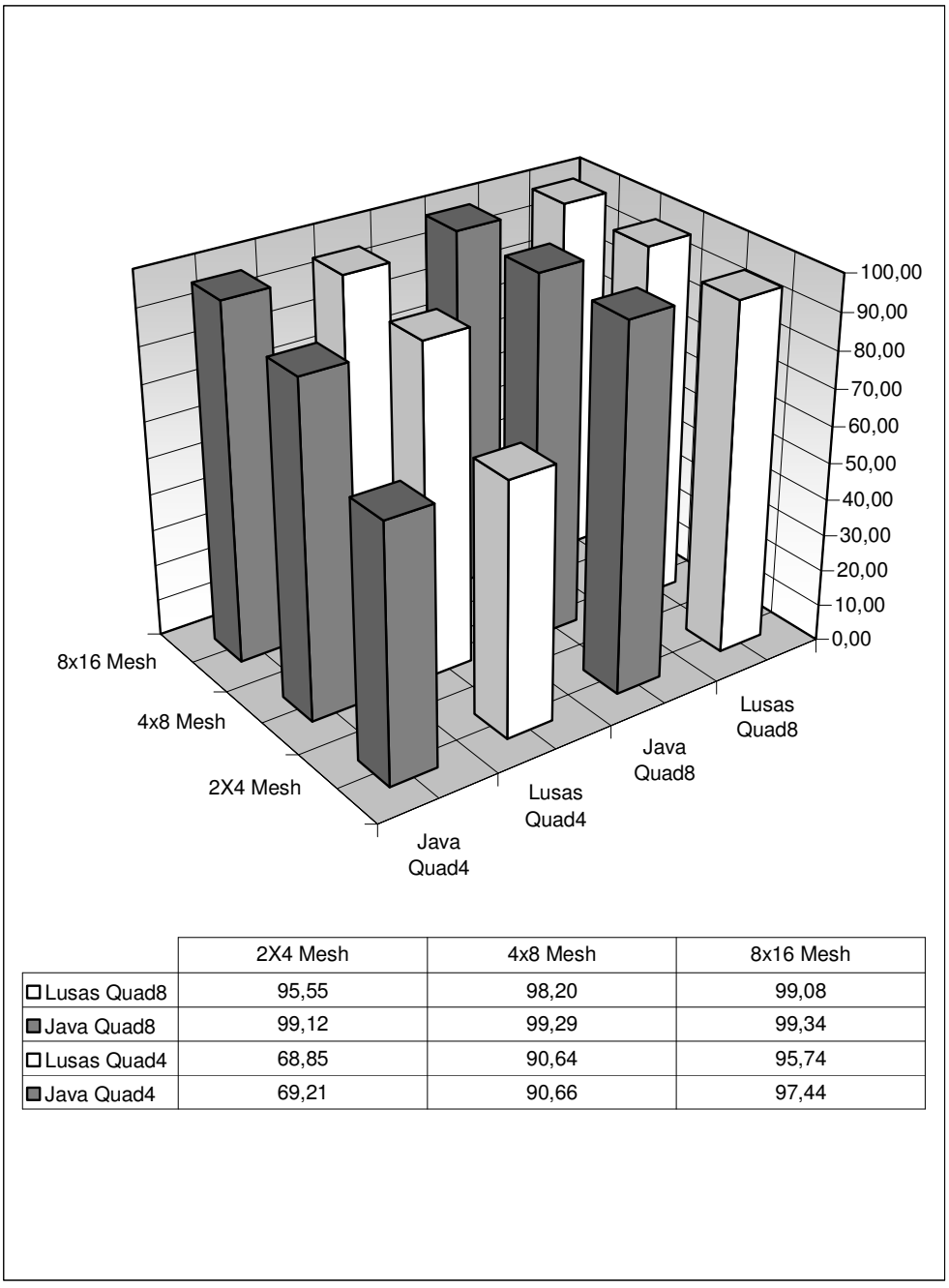
Table 9.17. Comparison of accuracies of normal stress at point E

Normal Stress(σ_x) Accuracy at x=7,y=2							
Name	**Exact Solution**	**Accuracy(%)**					
		2X4 Mesh		4x8 Mesh		8x16 Mesh	
		Lusas	Java	Lusas	Java	Lusas	Java
Pnt3Trg	2100	25.88	30.03	57.21	57.84	78.29	81.22
Pnt6Trg	2100	92.60	95.97	96.93	97.24	98.54	99.36
Pnt4Quad	2100	68.85	69.21	90.64	90.66	95.74	97.44
Pnt8Quad	2100	95.55	99.12	98.20	99.29	99.08	99.34



	2X4 Mesh	4x8 Mesh	8x16 Mesh
Lusas Trg6	92,60	96,93	98,54
Java Trg6	95,97	97,24	99,36
Lusas Trg3	25,88	57,21	78,29
Java Trg3	30,03	57,84	81,22

Figure 9.17. σ_x Accuracy of triangular elements at point E



	2X4 Mesh	4x8 Mesh	8x16 Mesh
Lusas Quad8	95,55	98,20	99,08
Java Quad8	99,12	99,29	99,34
Lusas Quad4	68,85	90,64	95,74
Java Quad4	69,21	90,66	97,44

Figure 9.18. σ_x Accuracy of quadrilateral elements at point E

10. PLANE STRESS CASE OF A SIMPLY SUPPORTED BEAM

10.1. Introduction

The purpose of this section is to compare Java plane stress program with the ANSYS finite element software. The plane stress element Pnt9Quad will be tested with exact and ANSYS solutions.

The cantilever thin plate sketched in Figure 10.1 is subjected to a concentrated vertical load "P" which is applied at point D.

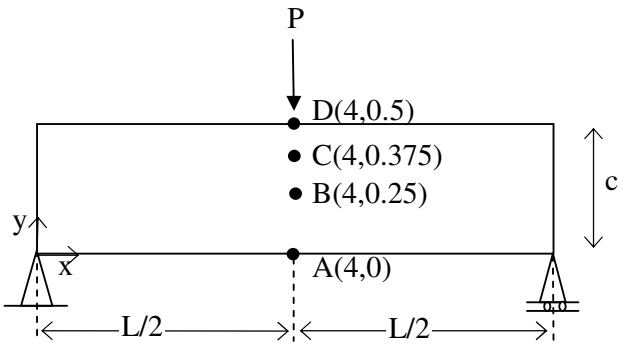


Figure 10.1. Simply supported beam plane stress problem

The material and geometric properties of the problem are given in the Table 10.1.

Table 10.1. Properties of the plane truss problem

Force (P)	20 kN
Length (L)	8 m
Thickness (t)	Unit
Depth (c)	0.5m
Elastic Modulus (E)	$3.\ 10^7$ kN / m^2
Poisson's ratio (ν)	0.3

10.2. Determination of the Exact Solutions

According to the theory of the elasticity, the exact values of stresses for points A, B, C, D and E are evaluated by the following formulas [18];

$$\sigma_x^{exact} = \sigma_x + \sigma'_x = \frac{3Pl}{4c^3} y + \beta_1 \frac{P}{c} \tag{10.1}$$

$$\sigma_y^{exact} = \beta_2 \frac{P}{c} \tag{10.2}$$

$$\tau_{xy}^{exact} = \beta_3 \frac{P}{c} \tag{10.3}$$

where I_{zz} is the moment of inertia and x and y are the local coordinates illustrated in Figure 10.2.

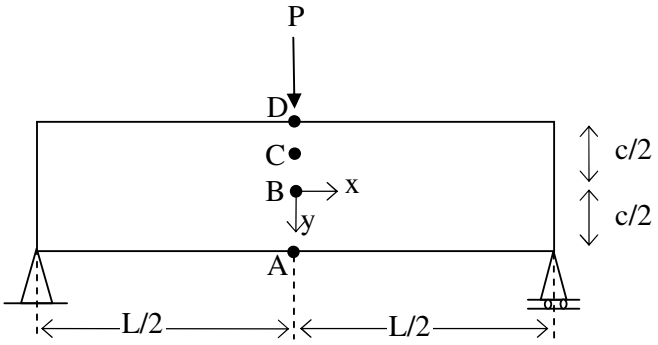


Figure 10.2. Local coordinates for exact solution formulas

The exact values of the vertical displacements along the middle axis of the beam can be evaluated by equation 10.4.

$$\delta_y = \frac{PL^3}{48EI} + \frac{PL}{4c}\left(\frac{3}{4G} - \frac{3}{10E} - \frac{3\nu}{4E}\right) - 0.21\frac{P}{E} \tag{10.4}$$

where ν is the poissons ratio and G is the shear modulus.

The numerical solutions of stresses and deflections at points A, B, C and D are given in the Table 10.2.

Table 10.2. The exact values of stresses

Point	β_i Values			Stress Calculation				
	β_1	β_2	β_3	σ_x	σ'_x	σ_x^{exact}	σ_y^{exact}	τ_{xy}^{exact}
A	-0.133	0.000	0.000	960.0	-10.6	949.4	0	0
B	0.121	-0.426	0.000	0.0	9.7	9.7	-36.5	0
C	0.428	-1.230	0.000	-480.0	34.2	-445.8	-98.4	0
D	N.A.	0.000	0.000	-960.0	N.A	N.A	∞	0

10.3. Data Entry

In the case of the plane stress problem introduced in Figure 10.1, mathematical model data of the simply supported beam is prepared by the template file given in the Appendix B.

Simply supported beam is subdivided into a fine mesh of 8 by 16. The coordinates of the nodes and corresponding node constrains are determined by Figure 10.1.

The modulus of elasticity matrix is obtained by the equation (9.4).

$$[E_{mat}] = \frac{E}{1-\upsilon^2}\begin{bmatrix} 1 & -\nu & 0 \\ -\nu & 1 & 0 \\ 0 & 0 & \dfrac{1-\nu}{2} \end{bmatrix} \tag{9.4}$$

The value of the point load and the node numbers of the elements are determined by the mesh figure illustrated in Figure 10.3.

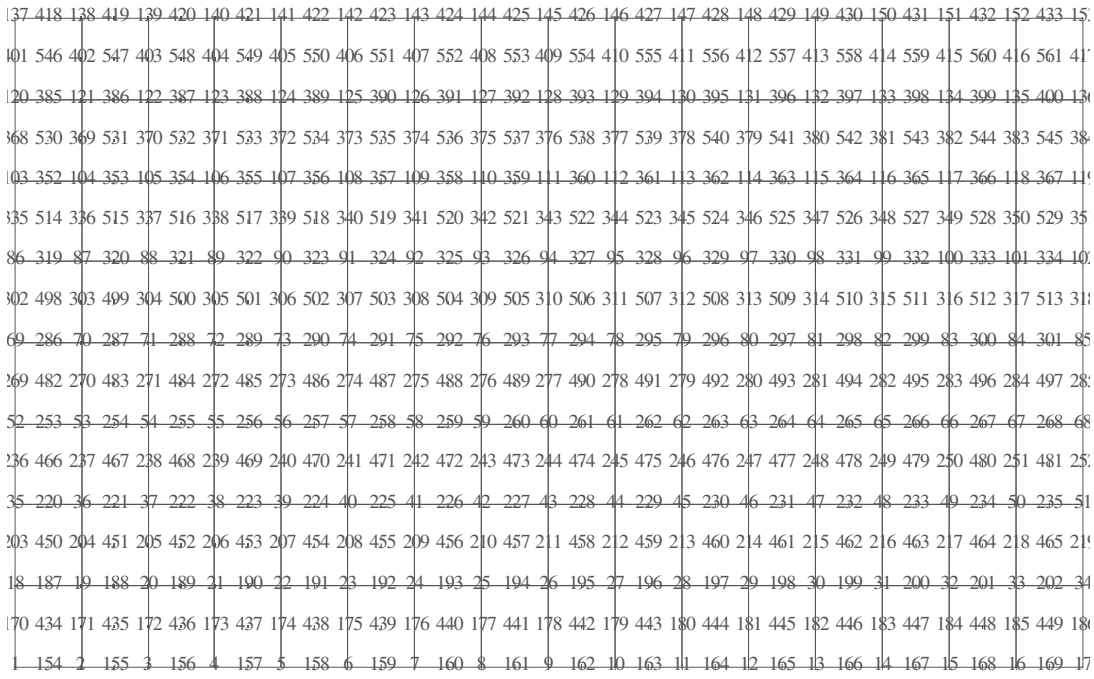


Figure 10.3. Quadrilateral mesh of simply supported beam case

10.4. Comparison of the Results with ANSYS Software

ANSYS is a general purpose finite element modeling package for numerically solving a wide variety of mechanical problems such as static/dynamic structural analysis heat transfer, fluid problems and acoustic and electro-magnetic problems.

The vertical displacements of the middle point B, the normal stress components of the points A, C, D and the shear stress of the A, B, C, D points are determined by running the Java program and the comparison tables are shown in Table 10.3.

The vertical displacements and horizontal displacements of the all element nodes are determined with Java program and the surface graphs are plotted in Figure 10.5 and Figure 10.7, respectively. In order to compare the performance of the Java program contour plots of the Ansys solutions are illustrated in Figure 10.4 to Figure 10.6.

Finally, the normal stress and shear stress components of the all element nodes are determined with Java program and the surface graphs are plotted in Figure 10.9 to Figure 10.14. In order to compare the performance of the Java program with Ansys software, contour plots of the Ansys solutions are illustrated in Figure 10.8 to Figure 10.13.

Table 10.3. Comparison tables of the stresses and deflections

Comparison of σ_x (kN/m^2)						
Point	y	σ_x^{exact}	JAVA		ANSYS	
			Result	Accuracy(%)	Result	Accuracy(%)
A	0.25	949.360	954.96	99.41	959.60	98.92
C	-0.125	-445.760	-477.73	92.83	-484.59	91.29
D	-0.25	∞	-1017.03	N.A.	-1016.90	N.A

Comparison of σ_y (kN/m^2)						
Point	y	σ_y^{exact}	JAVA		ANSYS	
			Result	Accuracy(%)	Result	Accuracy(%)
A	0.25	0.000	0.92	N.A.	1.33	N.A
C	-0.125	-98.400	-93.07	94.58	-65.90	66.97
D	-0.25	∞	-169.84	N.A.	-106.16	N.A

Comparison of τ_{xy} (kN/m^2)						
Point	y	τ_{xy}^{exact}	JAVA		ANSYS	
			Result	Accuracy(%)	Result	Accuracy(%)
A	0.25	0.000	0.00	100.00	0.00	100.00
B	0	0.000	0.00	100.00	0.00	100.00
C	-0.125	0.000	0.00	100.00	0.00	100.00
D	-0.25	0.000	0.00	100.00	0.00	100.00

Comparison of δ_y (10^{-5}m)						
Point	y	δ_y^{exact}	Java		ANSYS	
			Result	Accuracy(%)	Result	Accuracy(%)
B	0	6.90126667	6.9087	99.9	6.9213	99.7

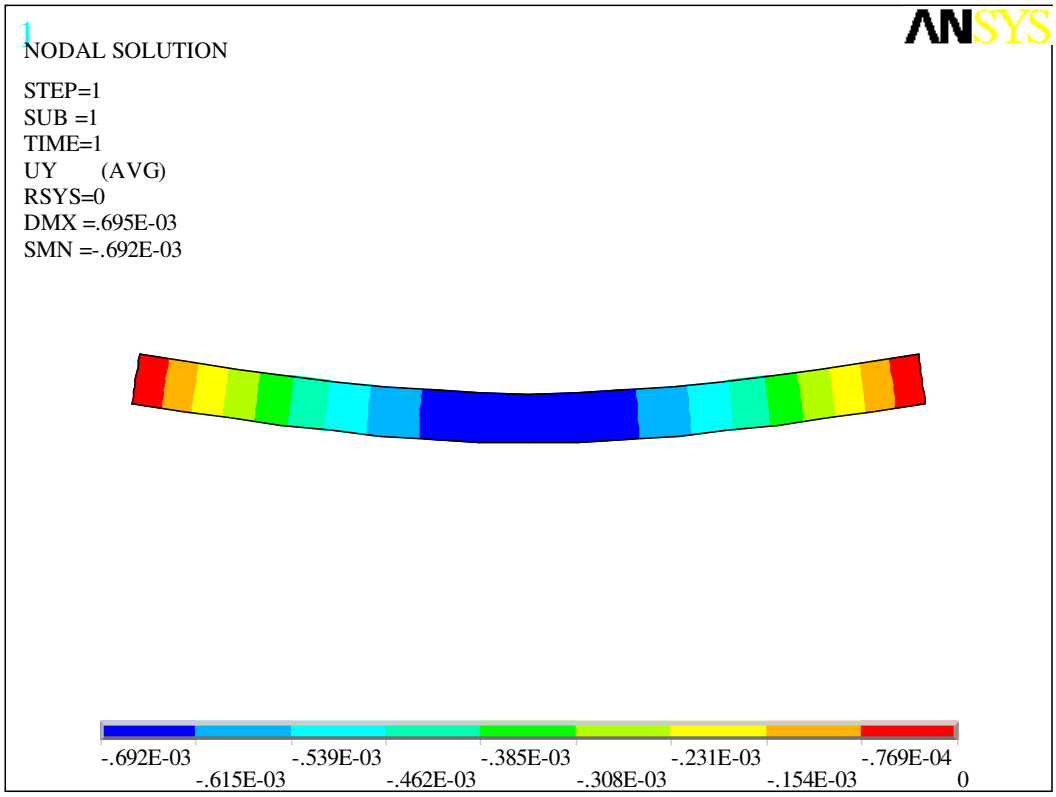


Figure 10.4. Contours of Ansys δ_y solution

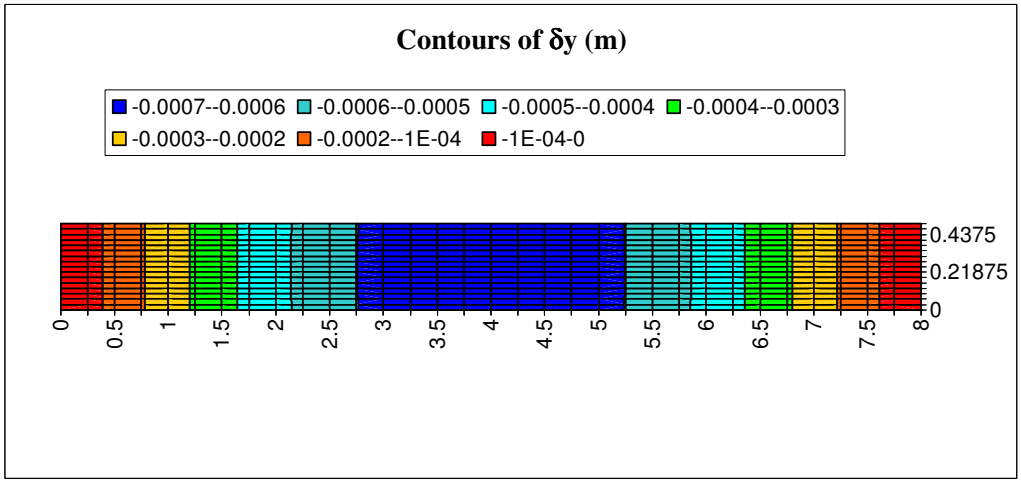


Figure 10.5. Contours of Java δ_y solution

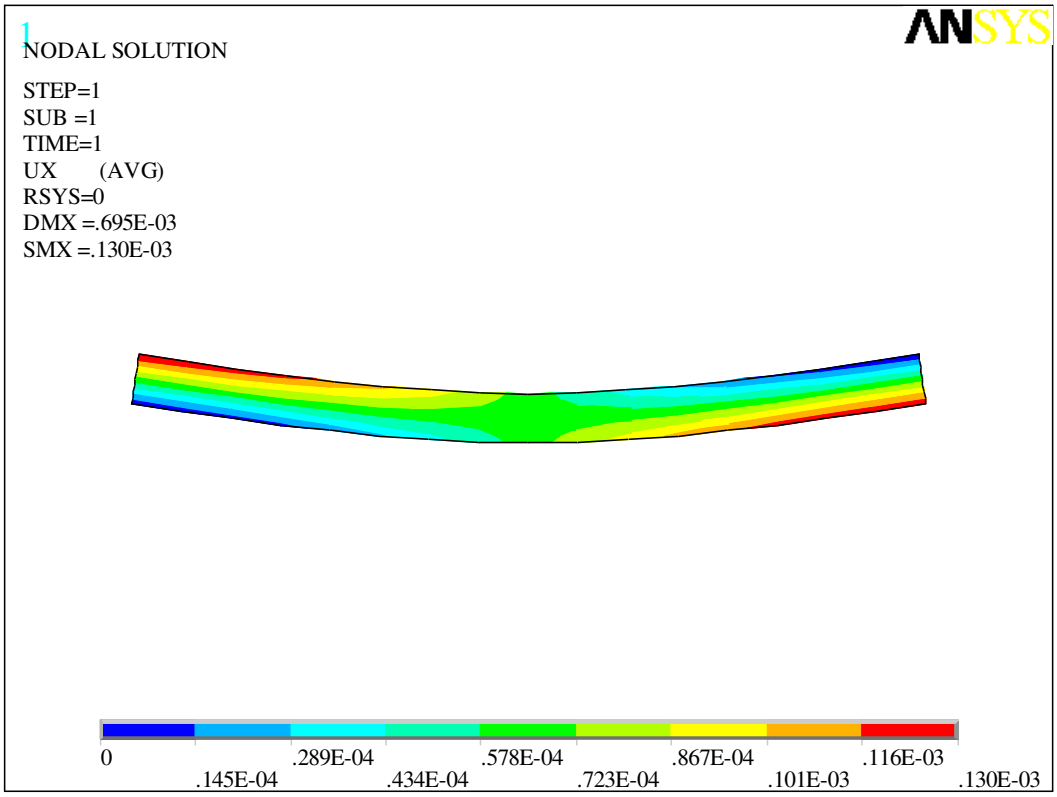


Figure 10.6. Contours of Ansys δ_x solution

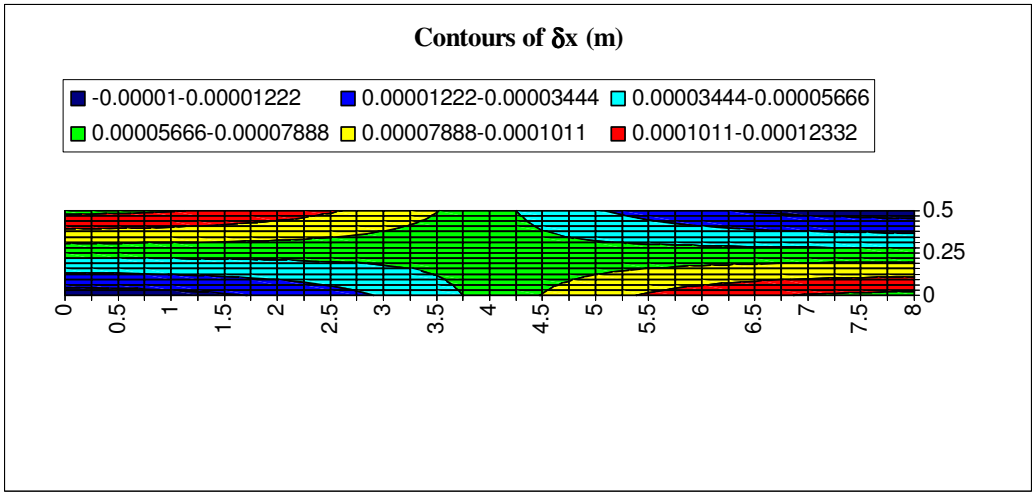


Figure 10.7. Contours of Java δ_x solution

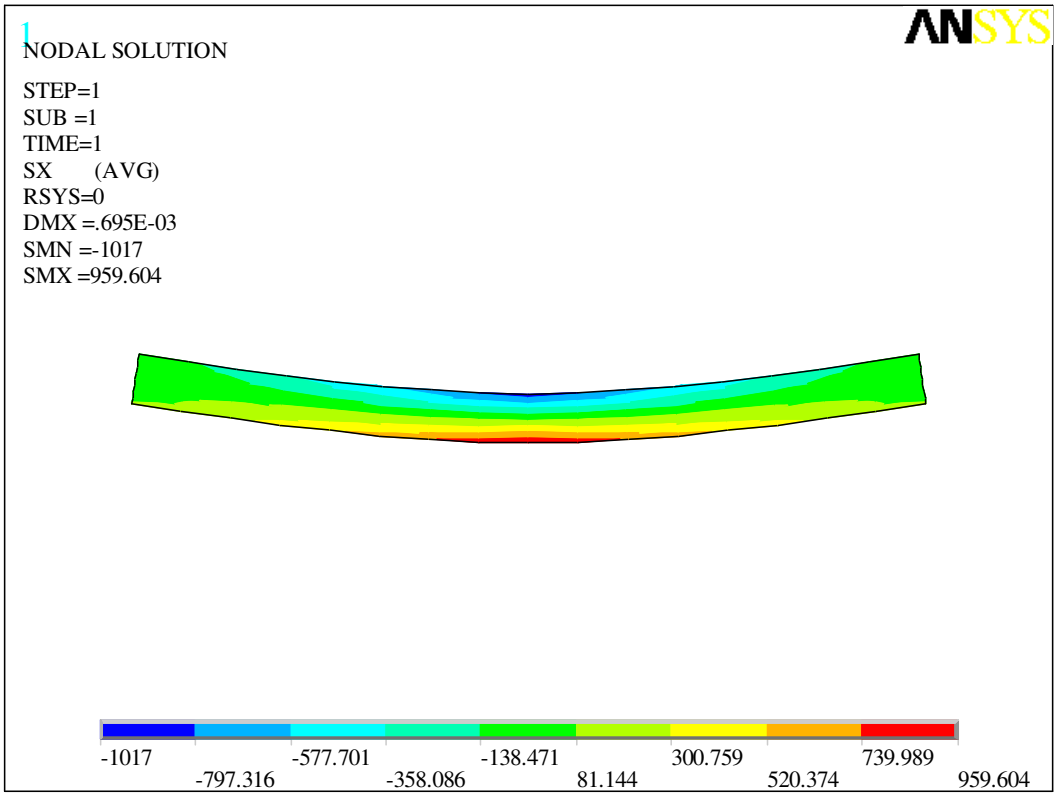


Figure 10.8. Contours of Ansys σ_x solution

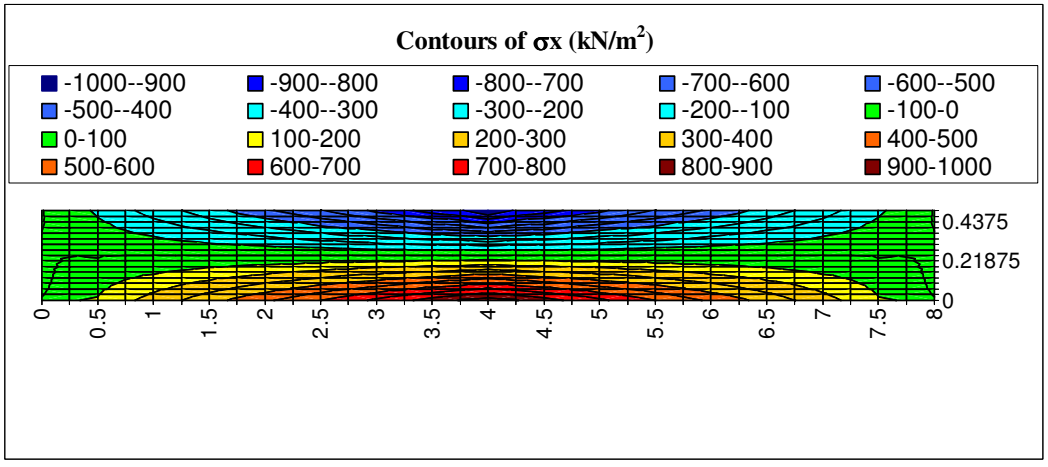


Figure 10.9. Contours of Java σ_x solution

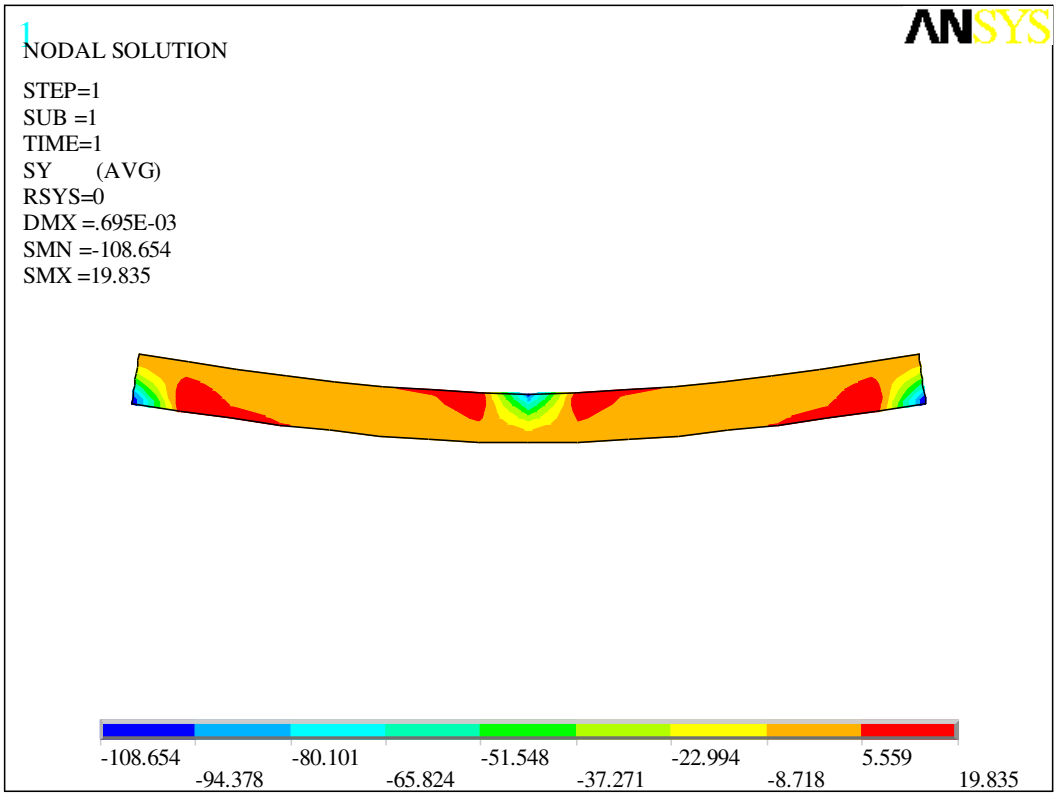


Figure 10.10. Contours of Ansys σ_y solution

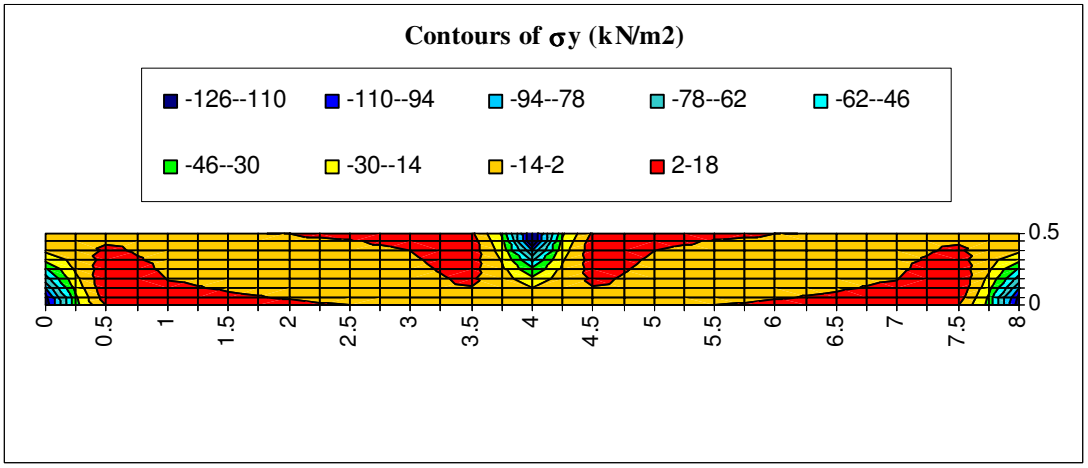


Figure 10.11. Contours of Java σ_y solution

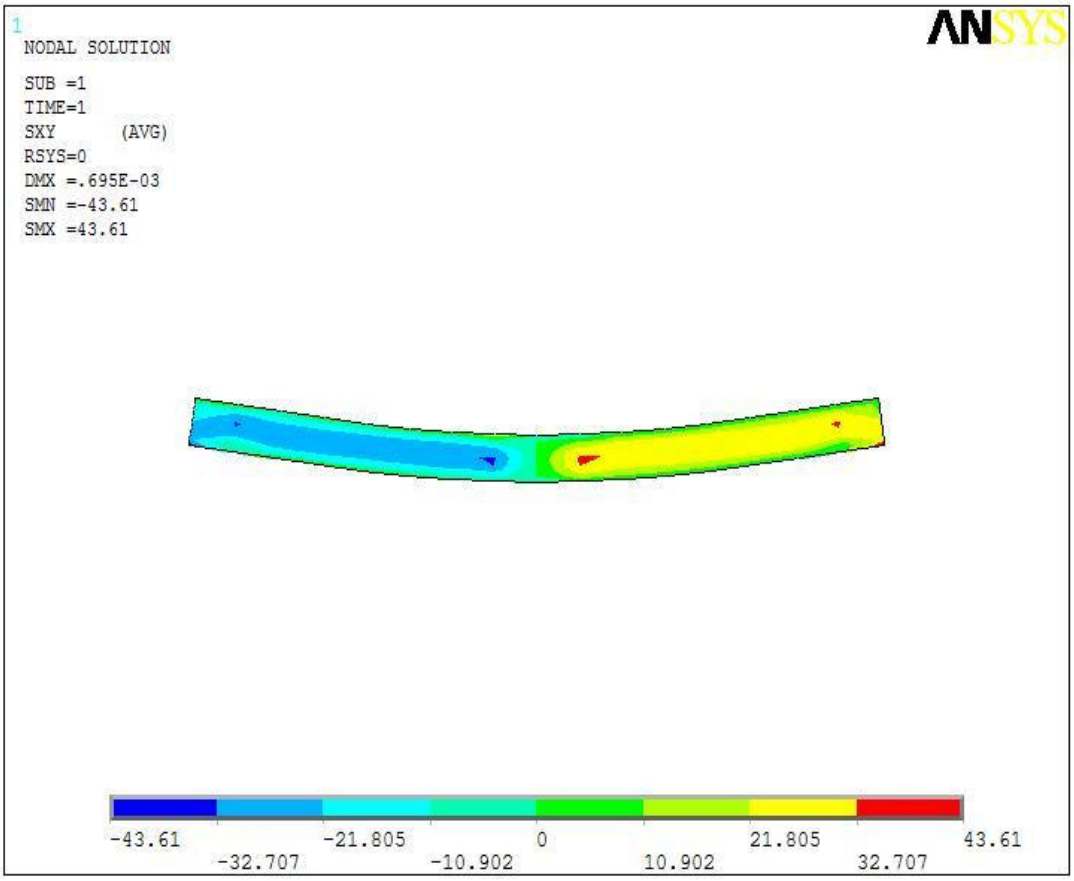


Figure 10.12. Contours of Ansys τ_{xy} solution

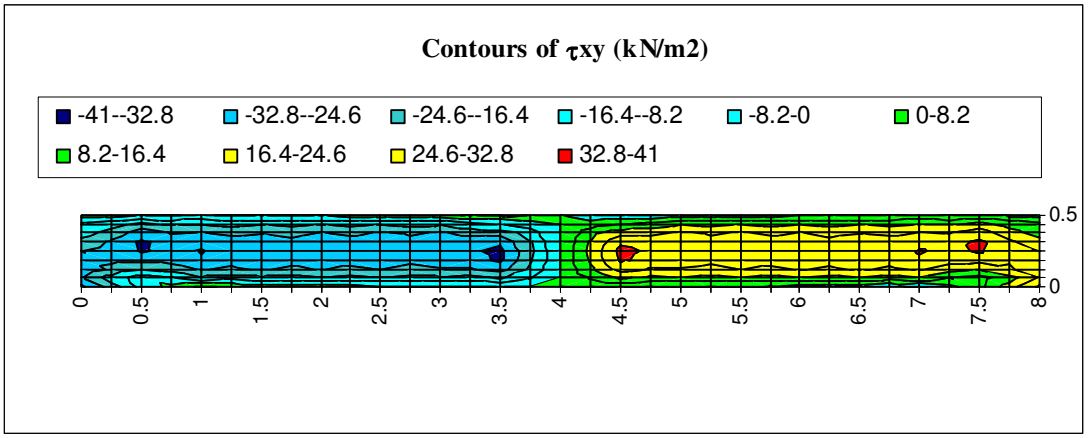


Figure 10.13. Contours of Java τ_{xy} solution

11. CONCLUSIONS

In this study, the architecture of one and two dimensional finite element programs written in the Java programming languages has been presented. The performance of the Java programs were compared with exact solutions and the commercial software packages.

On the basis of the algorithms presented herein, five different case studies were chosen to compare the applicability of the finite element codes. In the first three case studies, rod element and beam elements, in the forth case study, basic and higher order triangular and quadrilateral elements, finally in the fifth case study, nine node quadrilateral element were tested.

- 3DTruss and Beam-Column Frame programs are utilized with the three different structural problems. The results of the case studies indicate that the one dimensional finite element programs implemented in this study reproduce the exact solutions.

- Two dimensional Java plane stress finite element program is firstly utilized with the cantilever plane stress beam problem. As it is expected the results of the cantilever beam problem shows that basic triangular elements are the most incapable and inaccurate elements for the solution of the plane stress problems. The relatively higher accuracies are obtained when higher order finite elements are used. Besides, higher order elements approach the exact solution faster than the regular order elements.

- The comparison tables of the cantilever beam problem also indicate that for each element type, the validation of Java modules has been assessed with exact solutions resulting in excellent performances of two dimensional elements.

- Moreover, in the simply supported beam case with a concentrated load in the middle, the validation of Java nine node quadrilateral element module has been provided with exact solutions.

The general conclusion is that Java language allows to develop well-organized finite element applications. Therefore, the proposed Java programs can be acceptable and easily used in the several types of problems with superior computational efficiency.

Since the structure of the produced Java finite program is so practicable to develop that 3 dimensional element modules can be swiftly adapted to readily available interfaces introduced in this thesis through minor additions, omissions and modifications.

APPENDIX A – CONVERGENCE OF ANALYTIC RESULTS

Since the finite element method is a numerical procedure for solving the engineering problems, important considerations pertain to the accuracy of the analysis and the convergence of the numerical solution. The finite element method solution should converge to the analytic solution of the mathematical model which consists of Kinematics, Material, Loading and Boundary conditions.

In numerical analysis, the Lax equivalence theorem states, a consistent finite difference approximation for a well-posed linear initial value problem is convergent if and only if it is stable. In other words, consistency and stability are necessary and sufficient conditions for convergence. In order to assure "consistency" in finite element method, the elements must be "complete" and the elements and mesh must be "compatible". Stability, on the other hand, requires the system of finite element equations to prevent zero energy modes in elements as well as to preclude excessive element distortion.

A.1. Consistency

For the mathematical statement of the consistency, variational index, "m" which is the highest spatial derivative of displacement function appears in energy approach is to be considered. Since displacements are given in terms of first order derivatives of the displacements, variational index of the plane stress elements equals to one. The completeness of the plane stress problems requires that within each element, shape functions chosen as trial function, must contain a complete polynomial of degree m which equals to the variational index of one. Therefore, completeness is satisfied if the sum of the shape functions is unity for the isoparametric plane stress elements [12].

Moreover, the more important requirement is that compatibility of field variable must be maintained across element boundaries and this requires that shape function and its derivatives through order m-1 must be continuous across element edges. In plane stress, since strains are defined by first derivatives of the displacement fields, simple continuity of

the displacement fields across element edges suffices - this is called C^0 continuity. Compatibility between adjacent elements for problems which require only C^0 continuity can be easily assured if the displacements along the side of an element depend only on the displacements specified at all that nodes placed on that edge. Physically, compatibility ensures that no gaps occur between elements when the assemblage is loaded.

A.2. Stability

The requirement of "stability" of a finite element model means that the element stiffness matrices of the model must be able to represent the rigid body modes without excessive element distortion. Thus, stability is not a property of shape functions. Stability is required to be hold at the element level by obtaining rank sufficiency.

The element stiffness matrix must only produce zero internal energy in rigid body modes. Let n_F be the number of element degrees of freedom, and n_R be the number of independent rigid body modes. Let r denote the rank of K^e. The element is called rank sufficient if $r = n_F - n_R$ and rank deficient if $r < n_F - n_R$ [12].

For the isoparametric elements the rank sufficiency is defined by equation (A.1)

$$n_B n_G \geq n_F - n_R \tag{A.1}$$

where n_B is the order of the B (stress-strain) matrix and n_G is the number of Gauss points.

Since in the plane stress problem the stress-strain matrix is a 3x3 matrix of elastic modulus, n_B equals to three. Besides, independent rigid body modes are also three. Consequently, required Gauss points to achieve rank sufficiency are given in Table A.1.

Table A.1. Number of Gauss points for rank-sufficient plane stress elements

Element	n_F	n_R	n_F-n_R	n_E	n_G	Rule
Pnt4Quad	8	3	5	3	2	2x2 Gauss
Pnt8Quad	16	3	13	3	5	3x3 Gauss
Pnt9Quad	18	3	15	3	5	3x3 Gauss
Pnt3Trig	6	3	3	3	1	r=1 Gauss
Pnt6Trig	12	3	9	3	3	r=3,-3 Gauss

REFERENCES

1. Fish, J. and T. Belytschko, *A First Course in Finite Elements*, John Wiley & Sons, UK, 2007.

2. Encyclopædia Britannica, *Numerical Analysis*, Encyclopædia Britannica Online, http://www.britannica.com/eb/article-9108506, 2007.

3. Courant, R., "Variational Methods for the Solution of Problems of Equilibrium and Vibrations", *Bulletin of American Mathematical Society,* Vol. 49, pp. 1-23, 1943.

4. Ottosen, N. and H. Petterson, *Introduction to the Finite Element Method*, Prentice Hall, UK, 1992.

5. Rao, S. S., The Finite Element Method in Engineering, Pergamon Press, Oxford, 1989.

6. Liang, D.Y., *Introduction to Java Programming-Comprehensive Version*, Prentice Hall, New York, 2006.

7. Barker J., *Beginning Java Objects: From Concepts To Code*, Apress, USA, 2005.

8. Richard, A. J., *An Introduction to Java Programming and Object-Oriented Application Development*, Course Technology, USA, 2006.

9. Greenough, C., *The Finite Element Library*, Rutherford Appleton Laboratory, Oxfordshire, 2001.

10. Tezcan, S., *Lecture Notes on Finite Element Method*, Boğaziçi University, 2004.

11. Smith, I.M. and D.V Griffiths, *Programming the Finite Element Method*, John Wiley & Sons, UK, 2004.

12. Felippa, C., *Course Materials on Introduction to Finite Element Methods*, Colorado University, 2006.

13. Camp, C., *Development of Plane Stress and Plane Strain Stiffness Equations,* http://www.ce.memphis.edu/7117/pdf_notes/chapter06.pdf

14. Taig, I.C., *Structural Analysis by the Matrix Displacement Method*, Technical Report, Preston, 1961.

15. Ergatoudis, I. and Zienkiewics, O.C., *Curved, Isoparametric Quadrilateral Elements for Finite Element Analysis*, Int. J. Solids and Struct., vol. 4, 1968.

16. M. Abromowitz and I.A. Stegun, *Handbook of Mathematical Functions*, volume 55 of Applied Mathematics Series, National Bureau of Standards, Gathersburg, 1964.

17. Strang, G. and G. Fix, *Analysis of the Finite Element Method*, Prentice-Hall, Englewood Cliffs, 1973.

18. Timoshenko, S.P., *Theory of Elasticity*, McGraw-Hill, New York, 1934.

19. Timoshenko, S.P. and J.N. Goodier, *Theory of Elasticity*, McGraw-Hill, New York, 1970.

20. Harmandar, E., *Performance of Higher Order Finite Elements*, M.S. Thesis, Boğaziçi University, 2002.

REFERENCES NOT CITED

Altınbaş, B.A, *Java Yazılım Tasarımı*, Papatya, İstanbul, 2005.

Bate, K.J. and E.L. Wilson, *Numerical Methods in Finite Element Analysis*, Prentice Hall, Englewood Cliffs, 1976.

Berg, P.N., *Calculus of Variations*, McGraw-Hill, London, 1962.

Finlayson, B.A., *The Method of Weighted Residuals and Variational Principles*, Academic Press, New York, 1972.

Horne, M.R. and W. Merchant, *The Stability of Frames, Pergamon Press*, Oxford , 1965.

Jennings, A., *Matrix Computation*, John Wiley&Sons, Chichester , 1992.

Livelsley R.K., *Matrix Methods of Structural Analysis*, Pergamon Press, Oxford, 1975.

Strang, G. and G.J. Fix, *An Analysis of Finite Finite Element Method*, Prentice Hall, Englewood Cliffs, 1973.

Segerlind, L.J., *Applied Finite Analysis,* John Willey& Sons, New York, 1976.

Tezcan, S., *Lecture Notes* on Matrix Methods of Structural Analysis, Boğaziçi University, 2004.