Old Dominion University
ODU Digital Commons

Civil & Environmental Engineering Theses & Dissertations

Civil & Environmental Engineering

Spring 1998

Sparse Equation-Eigen Solvers for Symmetric/Unsymmetric Positive-Negative-Indefinite Matrices with Finite Element and Linear Programming Applications

Kakizumwami Birali Runesha
Old Dominion University

Follow this and additional works at: https://digitalcommons.odu.edu/cee_etds

Part of the Civil Engineering Commons, and the Mathematics Commons

Recommended Citation

Runesha, Kakizumwami B.. "Sparse Equation-Eigen Solvers for Symmetric/Unsymmetric Positive-Negative-Indefinite Matrices with Finite Element and Linear Programming Applications" (1998). Doctor of Philosophy (PhD), Dissertation, Civil & Environmental Engineering, Old Dominion University, DOI: 10.25777/ypat-1a61
https://digitalcommons.odu.edu/cee_etds/46

This Dissertation is brought to you for free and open access by the Civil & Environmental Engineering at ODU Digital Commons. It has been accepted for inclusion in Civil & Environmental Engineering Theses & Dissertations by an authorized administrator of ODU Digital Commons. For more information, please contact digitalcommons@odu.edu.

SPARSE EQUATION - EIGEN SOLVERS FOR SYMMETRIC/UNSYMMETRIC POSITIVE-NEGATIVE-INDEFINITE MATRICES WITH FINITE ELEMENT AND LINEAR PROGRAMMING APPLICATIONS

by

Hakizumwami Birali Runesha
B.S., October 1989, University of Kinshasa
M.S., May 1993, Old Dominion University

A Dissertation submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirement for the Degree of

DOCTOR OF PHILOSOPHY

CIVIL ENGINEERING

OLD DOMINION UNIVERSITY
(May 1998)

Approved by:

____
Duc T. Nguyen (Director)

____
Zia Razzaq (Member)

____
Chuh Mei (Member)

____
Jae Yoon (Member)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ABSTRACT

SPARSE EQUATION-EIGEN SOLVERS FOR SYMMETRIC/UNSYMMETRIC POSITIVE-NEGATIVE-INDEFINITE MATRICES WITH FINITE ELEMENT AND LINEAR PROGRAMMING APPLICATIONS.

Hakizumwami B. Runesha
Old Dominion University, 1998
Director: Dr. Duc T. Nguyen

Vectorized sparse solvers for direct solutions of positive-negative-indefinite symmetric systems of linear equations and eigen-equations are developed. Sparse storage schemes, re-ordering, symbolic factorization and numerical factorization algorithms are discussed. Loop unrolling techniques are also incorporated in the coding to enhance the vector speed. In the indefinite solver, which employs various pivoting strategies, a simple rotation matrix is introduced to simplify the computer implementation. Efficient usage of the incore memory is accomplished by the proposed " *restart memory management* " schemes. A sparse version of the Interior Point Method, IPM, has also been implemented that incorporates the developed indefinite sparse solver for linear programming applications.

Numerical performance of the developed software is conducted by performing the static analysis and eigen-analysis of several practical finite elements models, such as the EXXON Offshore Structure, the High Speed Civil Transport (HSCT) Aircraft, and the Space Shuttle Solid Rocket Booster (SRB). The results have been compared to benchmark results provided by the Computational Structural Branch at NASA Langley Research Center. Small to medium-scale linear programming examples have also been used to demonstrate the robustness of the proposed sparse IPM.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A ma Mère,
à mon Père,
à Didina, Veda, Rita, Manunu,
et à vous tous mes frères et soeurs,
pour votre amour et sollicitude, je vous dédie ce travail.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ACKNOWLEDGMENTS

I wish to express my sincere gratitude to my advisor, Prof. Duc T. Nguyen, for his valuable advice and his encouragement through the course of the research. I would also like to thank the other members of my dissertation committee, Prof. Zia Razzaq, Prof. Chuh Mei and Dr. Jae Yoon for their suggestions, comments and beneficial discussions.

The computer facility and financial support provided by the Computational Structural Branch at NASA Langley Research Center (several numerical data are provided by Dr. Olaf O. Storaasli) and ODU/CEE department are also acknowledged. The author also would like to acknowledge the source codes provided by Dr. Esmond G. Ng at Oak Ridge National Laboratory and by Dr. Ian Duff at Harwell Laboratory, which have been used in this study. Very helpful discussions with my colleagues, Dr. Chen Pu (former research associate at the Hong Kong University of Science and Technology) and Dr. J. Qin (former research associate at ODU) are also appreciated.

Finally, I would like to thank my family, miles away, for their support, love and encouragement.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE OF CONTENTS

Page

LIST OF TABLES xi

LIST OF FIGURES xvi

NOTATION xix

CHAPTER I INTRODUCTION 1

1.1 Overview 1

1.2 Review of previous work 2

1.3 Objectives and scope 8

CHAPTER II VECTOR SPARSE SOLVER FOR SYMMETRIC POSITIVE DEFINITE MATRICES 12

2.1 Introduction 12

2.2 Sparse storage for the coefficient stiffness matrix 14

2.2.1 Introduction 14

2.2.2 The sparse row-wise format 15

2.2.3 NASA Format 17

2.2.4 Fundamentals of sparse matrix technology 19

2.3 Vector-sparse Gauss elimination without pivoting 22

2.3.1 Review of LDL^T factorization algorithm 22

2.3.2 Flowchart of the vector-sparse LDL^T solver 24

2.3.3 Ordering for Gauss elimination: Symmetric matrices-MMD 26

2.3.4 Sparse symbolic factorization-SYMFA 26

2.3.5 Ordered and unordered representation-TRANSA 34

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Page

2.3.6 Vectorization and finding Master(or Super) Degree of freedom . . 34

2.3.7 Sparse numerical factorization with loop unrolling strategies 38

2.3.8 Forward and backward solution 46

2.3.9 Sparse matrix-vector multiplication with unrolling strategies 47

2.4 The Modified OakRidge sparse equation solver 49

2.4.1 Introduction 49

2.4.2 The OakRidge data format 50

2.4.3 Modification of the OakRidge solver 50

2.4.3 Reuse of data in fast memory: CACHE 52

CHAPTER III VECTOR SPARSE SOLVER FOR INDEFINITE MATRICES 54

3.1 Introduction 54

3.2 Symmetric Indefinite Systems-Pivoting Strategies 56

3.2.1 Introduction 56

3.2.2 Pivoting strategies 57

3.2.3 Weighted pattern matching strategy 60

3.2.4 Rotation matrix 63

3.2.5 Consecutive search strategy 64

3.3 Symmetric Indefinite Systems-Restarting 65

3.3.1 Simultaneous symbolic and numerical factorization 67

3.3.2 Partial reduction 68

3.3.3 Pivoting searching and ending partial reduction criteria 68

3.3.4 Data Management 70

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Page

3.3.5 Permutation 74

3.4 Forward reduction and back substitution 75

3.5 Reordering of indefinite systems 76

3.6 The modified MA27 sparse indefinite solver 77

3.6.1 Introduction 77

3.6.2 MA27 data format and control parameters 78

3.6.3 Modified MA27 solver: ODU-MA27 79

CHAPTER IV SPARSE SUBSPACE AND LANCZOS ITERATION FOR THE SOLUTION OF POSITIVE DEFINITE AND INDEFINITE SYSTEMS 80

4.1 Introduction 80

4.2 Subspace Iteration 81

4.2.1 Basic Subspace iteration algorithm 81

4.2.2 Subspace iteration step by step algorithm 83

4.2.3 Subspace iteration for positive definite systems: LDL^T 83

4.2.4 Subspace iteration for indefinite systems:ODU-HKUST/ODU-MA27 85

4.3 Lanczos Iteration 86

4.3.1 The Lanczos iteration algorithm 86

4.3.2 The Lanczos iteration step by step procedure 88

4.3.3 Lanczos iteration for positive definite systems: LDL^T 89

4.3.4 Lanczos iteration for indefinite systems: ODU-HKUST-ODU-MA27 . 90

4.4 Major computational tasks and enhancements in Subspace iteration and Lanczos algorithm 90

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Page

CHAPTER V INTERIOR POINT METHOD WITH POSITIVE AND INDEFINITE SPARSE SOLVERS FOR LINEAR PROGRAMMING PROBLEMS 92

5.1 Introduction 92

5.2 Review of the simplex method 93

5.3 Interior Point Method 96

5.3.1 Introduction 96

5.3.2 Variable transformation: Affine scaling method 98

5.3.3 Direction of move $\hat{C}_p$ 99

5.3.4 Step size σ 102

5.3.5 Feasible starting iteration vector $\vec{x}^o$ 103

5.4 Step by step algorithm for the IPM 105

5.5 Computational enhancements and the sparse implementation of IPM 106

CHAPTER VI VECTOR SPARSE SOLVER FOR UNSYMMETRICAL MATRICES 108

6.1 Introduction 108

6.2 Sparse storage of the unsymmetrical matrix 108

6.3 Basic unsymmetric equation solver 110

6.4 Vector-sparse LDU unsymmetrical solver 115

6.4.1 Introduction 115

6.4.2 Ordering for unsymmetrical solver 116

6.4.3 Sparse numerical factorization with loop unrolling 119

6.4.4 Forward and backward solution 120

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Page

6.4.5 Sparse unsymmetric matrix-vector multiplication 121

CHAPTER VII APPLICATIONS 123

7.1 Introduction 123

7.2 Description of various finite element models 125

7.2.1 Application No 1: High Speed Civil Transport (HSCT) Aircraft . 126

7.2.2 Application No 2: The EXXON Off shore Model 126

7.2.3 Application No 3: Thermal Structural Model 126

7.2.4 Application No 4: Solid Rocket Booster (SRB) 126

7.2.5 Indefinite matrices 127

7.2.6 Example descriptions for Interior Point Method 128

7.3 Numerical results 129

7.3.1 Sparse equation solvers 129

a)-LDL^T numfa1/2/8 129

b)-Cholesky OakRidgeODU solver 133

c)-ODU-HKUST indefinite solver 134

d)-ODU-Ma27 indefinite solver 136

7.3.2 Sparse eigen-solvers 136

a)-Subspace and Lanczos sparse eigensolvers for positive definite matrices 136

b)-Subspace and Lanczos sparse eigensolvers for indefinite matrices 139

7.3.3 Interior Point Method 141

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Page

7.3.4 Sparse unsymmetrical solver 142

CHAPTER VIII CONCLUSIONS AND SUGGESTIONS FOR FUTURE RESEARCH 235

8.1 Conclusions 235

8.2 Suggestions for future research 236

REFERENCES 238

APPENDICES 244

A. Multi-platform Makefile 244

B Subroutine cputime.f 247

C Parallel-Vector sparse solver 248

VITA 252

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF TABLES

TABLE Page

2.1 Solution time and storage requirement comparison for different storage schemes for a 263,574 degree of freedom FE Car model. 16

2.2 Skeleton Fortran Code for LDL^T 23

2.3 Skeleton Fortran coding for loop unrolling 35

2.4 Algorithm for finding Master DOF 38

2.5 Pseudo Fortran Skeleton Code for sparse LDL^T factorization 40

2.6 Numerical Factorization: ICHAINL update 43

2.7 Pseudo Fortran Skeleton Code for Sparse LDL^T factorization with unrolling strategies. 44

2.8 Fortran Skeleton code for the Vector portion of NUMFA2/8 46

3.1 Pivoting strategy for sparse symmetric indefinite systems 58

3.2 Pivoting strategy for sparse symmetric indefinite systems with Pattern matching 63

3.3 Indefinite solver: Restarting procedure 66

3.4 Simultaneous symbolic and Numerical Factorization 67

3.5 Restart algorithm for symmetric indefinite solver 71

3.6 Forward reduction and back substitution 76

4.1 Step by step algorithm for starting iteration vector 82

4.2 Step by step basic Subspace Algorithm 84

4.3 Step by step basic Lanczos Algorithm 89

5.1 Step by step solution process for optimization 93

5.2 Simplex Tableau 94

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE Page

5.3 A step of the simplex method 96

5.4 Step by step algorithm for the IPM optimizer 105

5.5 IPM algorithm 106

6.1 Portion of Skeleton Fortran code of reordering of an unsymmetrical matrix . . 118

6.2 Pseudo FORTRAN Skeleton Code For Sparse LDU Factorization 119

6.3 Pseudo FORTRAN Skeleton Code For Sparse LDU Factorization With Unrolling Strategies 121

6.4 Unsymmetrical matrix-vector multiplication 122

7.1 Characteristics of the NASA High Speed Civil Transport Aircraft FEM 145

7.2 Characteristics of the TLP Flexjoint EXXON FEM 151

7.3 Characteristics of the Thermal-Structural FEM 153

7.4 Characteristics of the Solid Rocket Booster FEM 155

7.5 Characteristics of Indefinite matrices applications 162

7.6 HSCT FEM: Memory requirement for different reordering algorithms 164

7.7 HSCT FEM: Comparison of results using MMD and different level of loop unrolling on the IBM R6000/590 *Stretch* machine 166

7.8 HSCT FEM: Comparison of results using MMD and different level of loop unrolling on the Sun SPARC 20 *rhino* machine 167

7.9 HSCT FEM: Summary of results on the IBM RS6000/590 *stretch* machine . . 172

7.10 HSCT FEM: Summary of results on the Sun SPARC 20 *rhino* machine 173

7.11 EXXON Off-shore FEM: Comparison of results using MMD and different level of loop unrolling on the IBM RS6000/590 *Stretch* machine 174

7.12 Thermal-Structural FEM: Comparison of results using MMD and different level of loop unrolling on the IBM RS6000/590 *Stretch* machine. 175

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE Page

7.13 SRB FEM: Comparison of results using MMD and different level of loop unrolling on the IBM RS6000/590 *Stretch* machine 176

7.14 HSCT FEM: K.INFO for Numfa8 177

7.15 HSCT FEM: Output file of Numfa8 on the *stretch* machine 178

7.16 HSCT FEM: OakRigdeODU solver. Impact of cache size on the IBM RS6000/590 *stretch* machine using MMD 179

7.17 HSCT FEM: OakRigdeODU solver. Impact of loop unrolling level on the IBM RS6000/590 *stretch* machine using MMD and cache size 64 180

7.18 HSCT FEM: OakRigdeODU solver. Impact of cache size on the IBM RS6000/590 *rhino* machine using MMD and loop 8 181

7.19 HSCT FEM: OakRigdeODU solver. Impact of loop unrolling level on the Sun SPARC 20 *rhino* machine using MMD and cache size 64 182

7.20 EXXON Off-shore FEM: OakRigdeODU solver. Impact of cache size on the IBM RS6000/590 *stretch* machine using MMD and loop 8 183

7.21 EXXON Off shore FEM: OakRigdeODU solver. Impact of loop unrolling level on the IBM RS6000/590 *stretch* machine using MMD and cache size 64 184

7.22 Thermal-Structural FEM: OakRigdeODU solver. Impact of cache size on the IBM RS6000/590 *stretch* machine using MMD and loop 8 185

7.23 Thermal-Structural FEM: OakRigdeODU solver. Impact of loop unrolling level on the IBM RS6000/590 *stretch* machine using MMD and cache size 64 186

7.24 SRB FEM: OakRigdeODU solver. Impact of cache size on the IBM RS6000/590 *stretch* machine using MMD and loop 8 187

7.25 SRB FEM: OakRigdeODU solver. Impact of loop unrolling level on the IBM RS6000/590 *stretch* machine using MMD and cache size 64 188

7.26 SRB FEM: K.INFO input file for OakRidgeODU solver 189

7.27 SRB FEM: OakRidgeODU solver. Output file on the *stretch* machine 190

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE Page

7.28 Percentage of Zero diagonal values of the Indefinite matrices 191

7.29 ODU-HKUST indefinite solver: Summary of results on Rhino 192

7.30 ODU-HKUST indefinite solver: Comparison of results on the Cray Y-MP .. 193

7.31 ODU-HKUST indefinite solver: Impact of using MMD and Zero-End on the *Stretch* machine 194

7.32 ODU-HKUST indefinite solver: Impact of the control paramater alpha on application No 9 (neq =15367) 195

7.33 ODU-Ma27: Summary of results on *Rhino* machine 196

7.34 ODU-Ma27: Summary of results on *stretch* machine 197

7.35 HSCT FEM: K.INFO for SPARSEPACK eigensolver 202

7.36 EXXON Off-shore FEM: " Sparse" Lanczos Algorithm from SPARSEPACK on stretch 203

7.37 EXXON Off-shore FEM: " Sparse" Subspace Iteration from SPARSEPACK on *stretch* 204

7.38 EXXON Off-shore FEM: Using Basic K.J. bathe's Subspace Iteration (KJBATHE96) on *stretch* 205

7.39 Application No 6: Subspace iteration for indefinite systems 206

7.40 Application No 6: Lanczos iteration for indefinite systems 207

7.41 K.INFO input file for the Lanczos and Subspace eigensolver for indefinite systems. 208

7.42 Application No 5: Lanczos for Indefinite systems on *Cedar* machine 209

7.43 Jonathan's ill-conditionned problem: NASA Langley test bed results. 210

7.44 Jonathan's ill-conditionned problem: Lanczos on *rhino* machine. 211

7.45 Johnathan's ill-conditionned problem: Subspace on *rhino* 212

7.46 NGST Satellite model (5156 DOF eigenproblem): Lanczos on *stretch* 214

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

TABLE Page

7.47 IPM: Small scale examples (for validating purposes) 216

7.48 IPM: Medium-scale examples (for timing purposes) on *Cedar* Sun workstation 217

7.49 K.INFO input file for the IPM 223

7.50 Application 16: Output file of IPM on *cedar* 224

7.51 HSCT FEM: Memory requirement for UNSYNUMFA 225

7.52 HSCT FEM: Summary of results for UNSYNUMFA1/2/8 using UnsyMMD and different level of loop unrolling on the IBM RS6000/590 *Stretch* machine ... 227

7.53 HSCT FEM: Comparison of results for UNSYNUMFA with no UnsyMMD and different level of loop unrolling on the IBM RS6000/590 *Stretch* machine ... 228

7.54 PierrotHSCT: Summary of results for UNSYNUMFA with UnsyMMD and different level of loop unrolling on the IBM RS6000/590 *Stretch* machine 231

7.55 SRB FEM: Summary of results for UNSYNUMFA using UnsyMMD and different level of loop unrolling on the IBM RS6000/590 *Stretch* machine 233

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LIST OF FIGURES

FIGURE Page

2.1 263,574 degree of freedom FE Car model 16

2.2 Flowchart of the Vector-sparse LDL^T solver 25

2.3 Master Degree of freedom 38

2.4 Numerical Factorization: Reduction of Row I by row L 41

2.5 Numerical Factorization: Update location of IUC of row j 42

2.6 Numerical Factorization: Loop unrolling 45

2.7 Sparse Matrix-Vector multiplication with unrolling strategies 48

2.8 Flowchart of the OakRidgeODU solver 51

3.1 Indefinite solver: Pivoting strategy 58

3.2 Indefinite solver: Pattern matching 62

3.3 Indefinite solver: Restarting procedure 67

3.4 Indefinite solver: Ending Partial reduction zone 69

3.5 Indefinite solver: Memory allocation of IWORK 70

3.6 Indefinite solver: Memory reallocation 72

3.7 Indefinite solver: Fill-in minimization 77

5.1 Effect of the initial point on the step length 97

5.2 Projective steepest ascent direction 100

6.1 Storage scheme for unsymmetrical matrix 109

6.2 Unsymmetrical solver: Factorization of u_{ij} and l_{ji} 113

7.1 High Speed Civil Transport aircraft, HSCT 144

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

FIGURE Page

7.2 Non-zero pattern of the NASA High Speed Civil Transport Aircraft FEM . . . 146

7.3 TLP Flexjoint Geometry Parameters of the EXXON FEM 147

7.4 A 3-D model of the TLP Flexjoint EXXON FEM 148

7.5 Schematic diagram of the TLP Flexjoint EXXON FEM 149

7.6 Non-zero pattern of the TLP Flexjoint EXXON FEM 150

7.7 Non-zero pattern of the Thermal-Structural FEM 152

7.8 FEM of the Solid Rocket Booster, SRB 154

7.9 Nonzero pattern of the FEM Solid Rocket Booster, SRB 156

7.10 Nonzero pattern of Application No 5-Cantilever beam problem 157

7.11 Nonzero pattern of Application No 6-Carlos Davilla Problem 158

7.12 Nonzero pattern of Application No 7-Jonathan's plate problem 159

7.13 Nonzero pattern of Application No 8-Knight's Panel problem 160

7.14 Nonzero pattern of Application No 9-15,367 DOF indefinite problem 161

7.15 Mcdonell Douglas Stitched/RFI All composite wing finite element model . . . 163

7.16 HSCT FEM: Non-zeros elements after factorization for different reordering schemes 165

7.17 HSCT FEM: Performance of Numfa1/2/8 on the *stretch* machine 168

7.18 HSCT FEM: Performance of Numfa1/2/8 on the *rhino* machine 169

7.19 HSCT FEM: Performance of Numfa8 for different compiler optimization level on the *stretch* machine 170

7.20 SRB FEM: Performance of Numfa8 for different compiler optimization level on the *stretch* machine 171

7.21 ODU-HKUST indefinite solver: Impact of the control parameter alpha on application No 9 (neq =15367) 195

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

FIGURE Page

7.22 HSCT FEM: Comparison of results for SPARSEPACK eigensolvers on *stretch* 198

7.23 HSCT FEM: Comparison of results for SPARSEPACK eigensolvers on *Rhino* 199

7.24 EXXON Off-shore FEM: Comparison of results for SPARSEPACK eigensolvers on *stretch* 200

7.25 EXXON Off-shore FEM: Comparison of results of SPARSEPACK eigensolvers on *USTSU31* 201

7.26 IPM: Graphical solution application No 11 218

7.27 IPM: Graphical solution application No 12 219

7.28 IPM: Graphical solution application No 13 220

7.29 IPM: Graphical solution application No 14 221

7.30 IPM: Graphical solution application No 15 222

7.31 HSCT: UNSYNUMFA. Non zero after factorization ($\times 10^6$) 226

7.32 HSCT FEM: Performance of UNSYNUMFA1/2/8 with UnsyMMD on the *stretch* machine 229

7.33 HSCT FEM: Performance of UNSYNUMFA1/2/8 with no UnsyMMD on the *stretch* machine 230

7.34 PierrotHSCT: Summary of results of UNSYNUMFA1/2/8 with UnsyMMD on the *stretch* machine 232

7.35 SRB FEM: Performance of UNSYNUMFA1/2/8 with UnsyMMD on the *stretch* machine 234

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

NOTATION

A_{ij} : ij element of matrix [A]

AD : diagonal values stored row by row before factorization

AN : non-zero off diagonal values stored row by row before factorization

$\vec{c}$: objective function

$\vec{C}_p$: direction of move

D : Diagonal matrix

DI : diagonal values stored row by row after factorization

$\vec{e}_i$: unit vector

f : Load vector

I : Identity matrix

IA : location in AN and JA of the first off diagonal value of each row before factorization

ICHAINL: Chained list

IU : location in UN and JU of the first off diagonal value of each row after factorization

JA : column indices of the non-zero off diagonal values stored row by row before factorization.

JU : column indices of the non-zero off diagonal values stored row by row after factorization.

K : Stiffness matrix

K^T : Transpose matrix of K

K^{-1} : Inverse matrix of K

L : Lower triangular matrix

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

M : Mass matrix

neq : Size (dof) of matrix K

ncoef : number of non-zero off diagonal values before factorization

ncoef2 : number of non-zero off diagonal values after factorization

P : permutation matrix

$(PAP^T)_{12} = a_{rp}$: matrix P permute row 1 with row r and row 2 with row p

R : rotation matrix

s : order of pivoting

T : Triadiagonal matrix

U : upper triangular matrix

UN : non-zero off diagonal values stored row by row after factorization

{x} : displacement vector

X_B : Basic variables

X_{NB} : Non basic variables

$\vec{X}^0$: Starting iteration vector

$\vec{X}^*$: optimum design

λ, μ : eigenvalues

ϕ, ψ : eigenvectors

ρ : shift value

σ : step size

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER I

INTRODUCTION[1]

1.1 Overview

The finite element method has been used successfully for the solution of many practical engineering problems in various disciplines, such as structural analysis, fluid mechanics, structural optimization, heat transfer etc. [1-5]. Essential to the finite element solution of these problems is an effective numerical procedure for solving large-scale, sparse systems of linear equations and generalized eigen-equations. These solution phases typically represent the most costly step of the analysis in terms of computational resources.

The solution of linear systems of equations on advanced parallel and/or vector computers is an important area of ongoing research [6-20]. The development of efficient equation solvers is particularly important for static and dynamic (linear and nonlinear) structural analysis, sensitivity analysis and structural optimization, control-structure interactions, ground water flows, panel flutters, eigenvalue analysis, heat transfer etc. [20-21]. Modern high-performance computers such as Cray-YMP, Cray-C90, Intel Paragon and IBM-SP2 have both parallel and vector capabilities; thus, algorithms that exploit these features are highly desirable.

On a single node computer processor with vector capability, it is generally safe to say that equation solvers based on sparse technologies are more efficient than ones based on the skyline and/or variable bandwidth technologies. Basic sparse equation solution

[1]The journal model used is: The International Journal of Numerical Methods in engineering, IJNM.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

algorithms have been well documented in the literature [10-11]. This is especially true for the cases where the coefficient matrix is symmetric and *positive definite*. However, for certain engineering applications, such as coupled analysis for structures with independently modeled finite element subdomains [21-23], optimization problems, nuclear reactor core modeling, circuit physics modeling, British gas pipe network distribution problem [8], the coefficient matrix is symmetric and *indefinite*. For these engineering applications, pivoting strategies are often required in order to avoid numerical difficulties during the LDL^T factorization process. Several pivoting strategies have been proposed in the literature [6-8,10]. These strategies, however, have been mostly developed and implemented for *dense* matrix. Only few promising sparse solvers with pivoting strategies, which can handle medium to large-scale indefinite system of equations, are available in the literature [8].

1.2 Review of Previous Work

For the past 20 years , while the performance of personal computers and workstations has increased tremendously, there has been an increasing interest in the use of computers with vector and parallel architecture for the solution of very large scientific computing problems. As a result of the impending implementation of such computers, there was considerable activity in the mid and late 1960's in the development of numerical methods. Some of these works were summarized in 1971 in the classical review article of Miranker [24]. It has only been in the period since then, however, that such machines have become available. The first supercomputer was put into operation at NASA's Ames Research Center in 1972, the same year that the first Texas Instruments Inc. Advanced Scientific Computer(TI-ASC) became operational in Europe, and the first Cray Research Inc. Cray-1 was put into service at Los Alamos National Laboratory in 1976. Since then, the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

supercomputers have evolved considerably. As computers grow in power and speed, matrices grow in size. In 1968, practical production calculations with linear algebraic systems of order 5000 were commonplace, while a "large" system was one of order 10 000 or more, [24]. Today, solving a quarter million system of equations on workstation is a common trend, [20]. A similar trend toward increasing size is observed in eigenvalue calculations.

The challenge for the numerical analyst is to devise the algorithms and arrange the computations so that the architectural features of a particular machine are fully utilized. Some of the best sequential algorithms that were unsatisfactory for large scale systems and needed to be modified or even discarded on sequential machines have had a rejuvenation because of new technologies such as sparse technology.

Traditionally, one of the most important tools for the numerical analyst to evaluate algorithms has been computational complexity analysis, i.e, operation counts. This arithmetic complexity remains an important tool for vector and parallel computers, but several other factors become equally significant. As we will see , vector computers achieve their speed by using an arithmetic unit that breaks a simple operation, such as a multiplication, into several subtasks, which are executed in an assembly line fashion on different operands. Two techniques for improving the performance of vector computers involve the restructuring of DO loops in Fortran in order to force a compiler to generate an instruction sequence that will improve performance. It is important to note that the underlying numerical algorithm remains the same. The technique of rearranging nested DO loops is done to help the compiler to generate vector instructions. The other technique, characterized as unrolling DO loops by Dongarra and Hinds in 1979 [24, 29], was initially used as a way to force the compiler to make optimal use of the vector registers on the Cray

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

computers. In its simplest form, loop unrolling involves writing consecutive instances of a DO loop explicitly with appropriate changes in the loop counter to avoid duplicate computation. Several examples were given by Dongarra in 1983 and Dongarra and Eisenstat in 1984, [24, 29], for basic linear algebra algorithms.

In many engineering applications, the most intensive numerical computation is the solution of systems of equations. These may arise, for example, in finite element procedure after the assembly. There have been numerous research works in the past two decades in the direct methods for solving linear systems of equations, mainly redesigning the Cholesky and Gaussian elimination algorithms with or without pivoting. Some of the issues considered were the storage scheme of the matrix, the ordering of the matrix, the vectorization technique, the ability to reuse data in cache, the amount of data movement, the memory access pattern and the pivoting strategies, just to cite a few.

The bulk of the work in Cholesky factorization of a symmetric positive definite matrix A occurs in a triply nested loop around the single statement

$$A_{ij} = A_{ij} - (A_{ik}A_{kj})/A_{kk} \tag{1.1}$$

By varying the order in which the loop indices i, j and k are nested, we obtain different formulations for the Cholesky factorization. The various versions of Cholesky factorization can be used to take better advantage of particular architectural features of a given machine (cache, virtual memory, vectorization, etc.) [25]. For more details concerning these versions of Cholesky factorization, consult George and Liu [30].

In some of todays finite element programs for large-scale applications profile matrix methods dominate. This category includes the skyline, variable band and frontal methods [10]. The characteristic feature of all these methods is that they only attempt to exploit zeros

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

in the finite element factor matrix outside a certain border. Inside the border, no attempts are made to exploit the zeros. Some attempts have been made to reduce the number of arithmetic operations, especially in connection with the variable band method. The main drawback of the envelope methods is their large storage requirements. This implies that out-of-core techniques are often necessary for large-scale systems.

The methods used for banded systems do not explicitly deal with the sparsity structure of the system. For banded matrices, this is not normally necessary because the matrix fills out to the band during the factorization. However, there are certain applications which produce very sparse matrices with little exploitable structure, and sparse arithmetic instructions play an important role. The idea is to store as vectors only the nonzero values, together with some arrays which indicate the locations of nonzero elements.

As noted by Duff in 1984 [10, 24, 27] , for example, the difficulty with vectorizing a general sparse routine is the indirect addressing. In order to avoid the problem of indirect addressing in sparse systems, Duff proposed using a frontal technique based on the variable band or profile scheme suggested by Jennings in 1976 [29]. The multifrontal method, introduced by Duff and Reid in [25, 27], is well documented in the literature. With much of its work performed within dense frontal matrices, this method has proven to be extremely effective on supercomputers [25]. Moreover, the multifrontal method is naturally expressed and implemented as a block method, and several of the advantages it derives from block matrix operations have already been explored in the literature: e.g., its ability to reuse data in fast memory and its ability to perform well on machines with virtual memory and paging [25].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

While the form of Gaussian elimination for dense matrix is an appropriate starting point for a new implementation, the architectural details of a particular machine may necessitate changes to achieve a truly efficient algorithm. Several early papers considered in great detail the implementation of Gaussian elimination and the Cholesky decomposition $A=LL^T$ on the first supercomputers. The variations of basic algorithms due to machine differences were summarized by Voigt in 1977 [24].

For banded systems, such as might arise from the discretizations of elliptic equations, the node points are ordered so as to achieve relatively small bandwidth. We now consider other orderings that are known to reduce both the number of arithmetic operations and the storage requirements for factoring the matrix of the resulting system. This is a very important issue in sparse matrix technology and constitutes a topic of research on its own. Most of the algorithms that minimize the fill-in are based on the graph theory. The most popular of these algorithms are the nested dissection and the minimum degree [30].

The most popular methods used in engineering practice for the solution of a few p eigenvalues and the associated eigenvectors of large finite element systems are the Subspace and Lanczos iteration methods. The Subspace iteration method developed and so named by K.J. Bathe, [1], consists of establishing q starting iteration vectors, ($q>p$), using simultaneous inverse iteration on the q vectors and Ritz analysis to extract the "best" eigenvalue and eigenvector approximations from the q iteration vectors. Altogether, the Subspace iteration method is largely based on various techniques that have been used earlier, namely, simultaneous vector iteration (F.L. Bauer and A. Jennings), Sturm sequence information, Rayleigh-Ritz analysis, and the work of H. Rutishauser [1]. Some advantages of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Subspace iteration are that the theory is relatively easy to understand and that the method is robust and can be programmed with little effort.

Lanczos algorithm for solving linear systems of equations and eigenproblems represent a very important computational innovation of the early 1950's. It became widely used only in the mid-1970's, [31]. Shortly thereafter, vector computers and massive computer memories made it possible to use this method to solve problems which could not be solved in any other ways. Since that time, the algorithms have been further refined and have become a basic tool for solving a wide variety of problems on a wide variety of computer architectures. Golub and O'Leary gave in their 1989 paper an extensive history of this method, [31]. In his work, C. Lanczos proposed a transformation for the tridiagonalization of matrices. However, as already recognized by Lanczos, the tridiagonalization procedure has a major shortcoming in the constructed vectors, which in theory should be orthogonal, but as a result of round-off errors, are not orthogonal in practice. A remedy is to use Gram-Schmidt orthogonalization, but such an approach is also sensitive to round-off errors and renders the process inefficient when a complete matrix is to be tridiagonalized. If the objective is to calculate only few eigenvalues and corresponding eigenvectors , the Lanczos iteration can be very efficient.

Karmarkar's publication in 1984 [32] of the new polynomial-time algorithm for linear programming drew enormous attention from the mathematical programming community and generated a lot of research activities during the past 13 years [33-35]. Soon after Karmarkar's publication, Gill and co-workers [33], have discovered that there is a close connection between this new (Karmarkar's) interior point method (or IPM) and the projected Newton Barrier methods. The IPM, in the earlier years could not be shown competitive to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the popular, unbeatable Simplex method [36], due to at least two reasons. First, due to the computer storage limitations, the size of the problems solved in the late sixties has been restricted to only a few hundred rows and columns, and for such small sizes, the simplex method is practically unbeatable. Secondly, it was only at the beginning of the seventies that a number of highly efficient sparse solvers have become available.

1.3 Objectives and scope

As with many other linear algebra algorithms, devising a portable implementation of a sparse solver that performs well both on the broad range of computer architectures currently available and for different type of problems is a formidable challenge. Even after limiting our attention to machines with only one processor, as we have done herein, there are still several interesting issues to consider. In this work we investigate sparse LDL^T Cholesky algorithms designed to run efficiently on vector supercomputers (e.g., the Cray Y-MP) and on powerful scientific workstations (e.g., the IBM RS/6000). To achieve high performance on such machines, the algorithms must be able to exploit vector processors. Moreover, with the dramatic increases in processor speed during the past few years, rapid memory access has become a very important factor in determining performance levels on several of these machines. To be efficient, algorithms must reuse data in fast memory (e.g., cache) as much as possible. Consequently, a highly localized and regular memory-access pattern is ideal for many of today's fastest machines. The cache size and the level of loop unrolling are machine-dependent parameters and are input values for the codes that we have developed.

The objective of this dissertation research can be summarized as follows:

- Review major existing profile and banded solvers and their out-of core implementation.
- Develop a robust vector sparse solver for positive definite matrix.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Develop new pivoting strategy, memory management and sparse solver for highly indefinite systems.

- Develop and implement a vector sparse Subspace and Lanczos procedure for positive, negative and indefinite systems.

- Review a version of Karmarkar Interior point Method.

- Develop a sparse version of interior point method by making use of the sparse technology and developed solvers.

- Develop a vector sparse unsymmetrical solver (unsymmetric in values but symmetric in locations).

- Solve practical structural analysis and optimization problems in order to evaluate the accuracy and speed of the developed procedures on different computer platforms.

This dissertation is organized into two parts. The first part consists of developing robust, efficient and fast solvers and the second part consists of making use of those solvers in developing efficient eigensolvers and IPM codes. After the introduction in Chapter I, Chapter II is devoted to developing a vector sparse solver for positive definite systems. Sparse storage schemes, symbolic factorization, re-ordering algorithms, numerical factorization, forward and backward solution strategies are discussed. Loop unrolling techniques are also incorporated into the sparse solver to enhance the vector speed. Modifications to the Cholesky Oak Ridge solver are also explained.

In Chapter III, a general purpose, robust and efficient (in terms of solution accuracy, memory requirements, and computational speed) sparse algorithm and the corresponding computer coding implementations for direct solution of indefinite system of linear equations are developed. The basic LDL^T algorithm for general symmetric coefficient matrix is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

reviewed. Extensions to the case where the symmetric coefficient matrix is *sparse* are discussed. An emphasis is put on the coding organization of the algorithm. Pivoting strategies for the proposed LDL^T algorithm for solution of sparse, symmetric and indefinite matrix are discussed. A restarting management scheme of the proposed algorithm is explained.

In Chapter IV, we re-examine the two popular eigen-solution algorithms: the Subspace and Lanczos iterations, incorporating recent developments in vectorized sparse technologies in conjunctions with Subspace and Lanczos iterative algorithms for computational enhancements. Basic Subspace iteration algorithm is reviewed. Key steps in Lanczos eigen-solution algorithm are summarized. Major computational tasks in Subspace and Lanczos iterative algorithms are identified and computational enhancements using vectorized, sparse strategies are discussed.

In Chapter V, a version of the interior point method is reviewed, and practical implementation of IPM is explained. Both the developed solvers for positive definite systems and indefinite systems are incorporated. The computational enhancements and the sparse implementation are explained.

In Chapter VI, a vector sparse solver for positive definite unsymmetric systems is developed. A special sparse storage scheme, modification to the reordering algorithm (MMD), numerical factorization for unsymmetric matrices and matrix-vector multiplication strategies are discussed. Vector unrolling in conjunction with the special sparse storage scheme is incorporated to enhance the vector speed.

In Chapter VII, several test problems have been conducted on different computer platforms in order to evaluate the numerical performance in terms of solution accuracy,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

memory requirements and computational speed of the proposed algorithms and their associated coding. Finally, conclusions and suggestions for future research are given in Chapter VIII.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER II

VECTOR-SPARSE SOLVER FOR SYMMETRIC POSITIVE DEFINITE MATRICES

2.1 Introduction

Let's consider the following system of linear equations

$$Kx = f \tag{2.1}$$

For many engineering applications, the coefficient matrix K often has nice properties, such as symmetry, positive definiteness and sparsity. Matrix K is *symmetric* when $K^T = K$, where T means transpose, i.e. when $K_{ij}=K_{ji}$ for all i and j. Otherwise K is unsymmetric. A symmetric matrix K is said to be *positive definite* when $y^T K y > 0$ for any vector y having at least one nonvanishing component. If two vectors y and z can be found for which $y^T K y > 0$ and $z^T K z < 0$, then A is said to be indefinite or nondefinite.

A square matrix L is *lower triangular* when it has nonzero elements only on or below the diagonal: $L_{ij} = 0$ if $i < j$ and some $L_{ij} \neq 0$ for $i \geq j$, with at least one $L_{ij} \neq 0$ for $i > j$. A lower triangular matrix is said to be *unit diagonal* if its diagonal elements are all equal to 1: $L_{ii} = 1$ for all i.

A square matrix U is *upper triangular* when it has nonzero elements only on or above the diagonal: $U_{ij} = 0$ if $i > j$ and some $U_{ij} \neq 0$ for $i \leq j$, with at least one $U_{ij} \neq 0$ for $i < j$. An upper triangular matrix is said to be *unit diagonal* if its diagonal elements are all equal to 1: $U_{ii} = 1$ for all i.

A necessary and sufficient condition for a symmetric matrix K to be positive definite is that the determinants of the n leading principal minors of K be positive. Also if K is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

symmetric positive definite, a unique Cholesky factorization $K=U^TU$ exists or $K = U'^TDU'$, where U is upper triangular with positive diagonal elements, D is diagonal with positive diagonal elements.

In equation (2.1), the vectors x and f represent the unknown nodal displacement and the known nodal load vectors, respectively. In general, matrix K can be factorized into either LDL^T or CC^T. In the LDL^T form, L is a lower triangular matrix with unit diagonal, and D is a diagonal matrix. In the CC^T form, CC^T is a non-negative matrix and C is a lower triangular matrix. The LDL^T form requires slightly more computational effort than the CC^T form. In several engineering applications, K is indefinite. In these cases, only the LDL^T form is applicable. Therefore, in our study, an emphasis is put on the LDL^T form for factorization. To solve a system of simultaneous equations, Eq. (2.1), three major steps are identified:

Step1: Factorization

$$K = LU = LDL^T \tag{2.2}$$

Step2: Forward reduction:

$$LDy = f \tag{2.3}$$

Step3: Back substitution:

$$L^Tx = y \tag{2.4}$$

In the above three steps, for a single right-hand vector, f, the factorization phase takes much of (more than 90%) the total computational time compared with the other two steps. Thus, improvements in solution efficiency should be focused on this part of the calculation. In some cases, such as the modified Newton-Raphson method for non-linear equation [1] and inverse subspace iteration for eigenvalue problems, [1], where the stiffness matrix K remains

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

constant for a number of load (or time) increments, computation steps (2) and (3), are employed repeatedly for different right-hand side vectors f. Therefore, for efficiency, improvements need to be considered on forward as well as back substitution.

2.2. Sparse storage for the coefficient stiffness matrix

2.2.1 Introduction

Direct methods for the solution of linear equations are equivalent to the factorization of the coefficient matrix. For large matrices, the optimization of the memory required to store the matrix as well as the arrays needed for the solution is as important as the efficiency of the algorithm. If only small number of equations is involved, then the factorized matrix can be stored as a full triangular matrix. However, when larger problems are encountered which do not fit into the machine storage or which involve redundant operations with a significant number of zero values, then other storage schemes become advantageous. Furthermore, to take advantage of the symmetry of the matrix, either the upper or lower part, is stored in the memory.

Many matrices have a banded structure, in that for every non-zero element a_{ij} of a matrix K we can calculate the difference $|i-j|$, and we call the largest of these the half bandwidth. This can be much smaller than the order of the matrix. It is only necessary to store the elements of the matrix within the band.

If the pattern of non-zero matrix elements is observed further, it is seen that the bandwidth of each row of the matrix is not affected by the Cholesky factorization process, although many elements within the band which are zero in matrix K become non-zero in L. This feature is exploited in the variable bandwidth storage scheme.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The storage saving achieved by adopting such schemes may still not be sufficient to store larger matrices in the memory of the machine used. Skyline storage scheme still contains a large proportion of zero elements. Thus, for better computational efficiency, one prefers to process and store only the non-zero elements under the skyline profile. There exist many types of storage format for sparse matrices. The next paragraph describes the format that has been used in all our computer coding implementation. To illustrate the benefits of using sparse technology, Table 2.1 compare the solution time and storage requirement for different type of storage schemes for a 263,574 degrees of freedom finite element car model [20].

2.2.2 The sparse row-wise format

The sparse row-wise format to be described is the most commonly used storage scheme for sparse matrices. The scheme has minimal storage requirements, and, at the same time, it has proved to be very convenient for several important operations such as addition, multiplication, permutation and transposition of sparse matrices, the solution of linear equations with sparse matrix of coefficients by either direct or iterative methods, etc. In this scheme, the values of the non zero elements of the matrix are stored by rows, along with their corresponding, column indices, in two arrays, say AN and JA, respectively. An array of pointers IA(1:neq+1), is also provided to indicate the starting locations in AN and JA where the description of each row begins. An additional array, AD(1:neq) is used to store the diagonal entries. Here, neq is the order of matrix K and ncoef is the total number of non-zero off-diagonal elements in the upper triangular matrix K. The dimension of arrays AN, JA is ncoef. Similarly, the factorized matrix is stored in four arrays UN(1:ncoef2), IU(1:neq+1), JU(1:ncoef2) and DI(1:neq) where ncoef2 is the number of non-zeros after factorization

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Solver Type	Full, Unsymmetrical	Banded, Symmetrical	Sparse
Storage Scheme	Full	Variable Band	Sparse
Memory Required	neq^2=6.97 10^{10} words	894,427,805 words	88,500,000 words (ncoef=6,267,099)
Total Solution Time	3407 Hours	Out-of-core:2,789sec Using 8 processors: 298 sec	100sec (-Reordering=44sec -Numerical Factorization= 43 sec)

Table 2.1 Comparison of solution time and storage requirements for different storage schemes on a 263,574 dof car model

Fig. 2.1 263,574 degree of freedom Car Model
(source: NASA Langley, Hampton Va)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To facilitate the discussions in this section, as an example, let's assume the coefficient matrix K takes the following form

$$K = \begin{bmatrix} 11. & 0. & 0. & 1. & 0. & 2. \\ & 44. & 0. & 0. & 3. & 0. \\ & & 66. & 0. & 4. & 0. \\ & & & 88. & 5. & 0. \\ & SYM & & & 110. & 7. \\ & & & & & 112. \end{bmatrix} \qquad (2.5)$$

In the sparse row-wise storage representation, the data in Eq. (2.5) can be represented as follows:

IA(1:7=*neq*+1) = {1, 3, 4, 5, 6, 7, 7 }

JA(1:6=*ncoef*) = {4, 6, 5, 5, 5, 6}

AD(1:6=*neq*) = {11., 44., 66., 88., 110., 112.}

AN(1:6=*ncoef*) = {1., 2., 3., 4., 5., 7. }

where *neq*: the size of the original stiffness matrix and

ncoef: the number of non-zero, off diagonal terms of the original stiffness matrix.

2.2.3. NASA Format

The data format of NASA benchmark sparse matrices is a set of six files (or seven files for eigen-problems) in ASCII format given as follows:

K.INFO : Contains number of equations and coefficients.

{n1, n2, n3, NEQ, NEQ, NCOEF, n7, n8, n9, n10 }

K.DIAG : Contains diagonal terms.

K.PTRS : Contains number of non-zero off-diagonal terms in each row.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

K.RHS : Contains right hand side (load vector).

K11.INDXS : Contains column number for each non-zero off-diagonal term.

K11.COEFS : Contains the real, numerical value of each non-zero off-diagonal term (in row-wise format).

K.DMASS : Contains the diagonal terms of the mass matrix

For eigenanalysis problems with consistent mass, an additional file, K.CMASS, is also provided that contains the off-diagonal terms of the mass matrix, with an assumption that the mass matrix has the same column indices structure as the stiffness matrix.

Let's consider the system of equations given in Eq.(2.1), with the stiffness matrix K given in Eq.(2.5) and a load vector {f}=[201, 202, 203, 204, 205, 206]. The input data in NASA format will be given as follows:

K.INFO = { 0, 0, 0, 6, 6, 6, 0, 0, 0, 0 }

K.PTRS = { 2, 1, 1, 1, 1, 0}

K11.INDXS = {4, 6, 5, 5, 5, 6}

K.DIAG = {11., 44., 66., 88., 110., 112.}

K11.COEFS = {1., 2., 3., 4., 5., 7. }

K.RHS = {201, 202, 203, 204, 205, 206}

In the coding implementation of the sparse solver, the input data is read either as ASCII or binary files in NASA format and the arrays K.PTRS is directly converted into an array of pointers IA that indicate the starting nonzero location in K11.COEFS and K11.INDXS of each row.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.2.4 Fundamentals of sparse matrix technology

In this section we introduce some terms and techniques used in sparse matrix technology related to the symbolic and numerical processing of sparse matrix, that we will frequently use in this research work.

a) Merging sparse lists of integers

Merging is equivalent of using " OR " in Fortran symbols. By merging two or more sparse lists, a new list is obtained. An integer belongs to the resulting list if and only if it belongs to any of the given lists, and no repeated integers are allowed. This operation of merging lists of integers is very important in sparse matrix technology because it is commonly used to form the list of the column indices associated with each of the rows of a new matrix, obtained by performing algebraic operations on another matrix or matrices particularly when sparse formats are used. Examples are addition, multiplication and triangular factorization of sparse matrices. The following example illustrates the concept of merging. Given these three lists:

list A : 2, 5, 3, 9

list B : 3, 11, 9

list C : 5, 2

the resulting merged list, say M, is:

merged list M : 2, 5, 3, 9, 11

The merged list is obtained by inscribing each integer from each of the given lists, provided the integer was not previously inscribed. In order to determine efficiently whether an integer was previously inscribed or not, we use an array, often called *expanded* array or *switch* array, say *ISWITCH*, where a conventional number, the switch, is stored at position i immediately

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

after the integer i has been added to the merged list M under construction. Conversly, before adding an integer k to the list M, we check whether the value stored in *ISWITCH*(k) is equal to the switch or not, and we add k only if it is not.

b) The multiple switch technique

Each time a merging operation starts, the switch array *ISWITCH* just discussed should not contain the switch value in any of its positions. This can be achieved by initializing the array *ISWITCH* to zero at the beginning and by using a positive integer as the switch.

However in sparse matrix technology, merging operations are used to construct the lists of column indices for say, the *neq* rows of a *neq* x *neq* matrix. In this case *neq* different merging operations are required for this purpose, all of them to be performed using the same array *ISWITCH* of length *neq* as the switch array. Gustavson, [20], suggested we set to 0 the *neq* positions of *ISWITCH* only once, and then we perform the *neq* merging operations using each time a different value for the switch parameter. The rule of thumb is to use 1 as the switch for the first merging operation, 2 for the second , and so on. In this way, when the first merging operation is started, all positions of *ISWITCH* contain 0. When the second one is started, all positions of *ISWITCH* contain either 0 or 1, which does not conflict with the use of 2 as the switch, and so on. Now, *neq* executions of the sentence *ISWITCH*(i)=0 are required for *neq* merging operations. There is an average of only one execution of *ISWITCH*(i)=0 for each merging operation. The multiple switch technique is also known as the *phase counter* technique, [20].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

C) Expanded real accumulator

One considers a row or a column of sparse matrix, only the numerical values of nonzeros are stored in the computer memory in a real array, say *RN*, and their corresponding column numbers in an integer array , say *JR*. Both arrays are of the same length, which is much smaller than *neq*. This storage of a vector by considering only nonzero values is said to be *compact* or *packed*. The numerical value of the nonzeros of the sparse vector can also be stored in an expanded form in a real array of length *neq* , say *X*, as if it were full vector. The column numbers, however, are stored in the array *JR* as before for the nonzeros values only. This type of storage is used only temporarily, usually during the execution of a program and when certain algebraic operations are to be performed on the vector. The existence of the array *JR* allows the algorithm to operate directly on the nonzeros and to keep the operation count much smaller than *neq*. In merging lists in the addition of two matrices A (IA, JA, AN) and B (IB, JB, BN) for example, a symbolic phase is first performed to determine the positions of the nonzeros or structure of the resulting matrix C (IC, JC, CN). Knowing the positions of the nonzeros in C (JC), the numerical section of the algorithm is used to determine their numerical values. This process is not straightforward. If for example column 2 is the first column number of JC, we will not try to find that index in JA and JB before the summation, instead we use an expanded storage of the vectors in an expanded array of dimension neq, say X, often called *expanded real accumulator*. Finally we retrieve the nonzeros numbers from X to form CN by using the array of column number, JC, to find where the nonzeros values are stored in X.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.3 Vector-Sparse Gauss Elimination (LDL^T) without pivoting

In this section, major building blocks for the development of the basic sparse algorithms without pivoting are summarized. The "unrolling" strategies for better performance on vector computers is also explained.

2.3.1 Review of LDL^T Factorization algorithm

The Cholesky (or U^TU) factorization is efficient, however its application is limited to the case where the coefficient stiffness matrix [K] is symmetrical and positive definite. With negligible additional computational efforts, the LDL^T algorithm can be used for broader applications (where the coefficient matrix can be either positive, or negative definite). In this algorithm, the given matrix [K] in Eq.(2.1) can be factorized as

$$[K] = [L]\,[D]\,[L]^T \tag{2.6}$$

where [L] and [D] are lower triangular matrix (with unit values on the diagonal) and diagonal matrix, respectively. For a simple 3x3 symmetrical stiffness matrix, Eq.(2.6) can be explicitly expressed as

$$\begin{bmatrix} K_{11} & K_{12} & K_{13} \\ K_{21} & K_{22} & K_{23} \\ K_{31} & K_{32} & K_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \begin{bmatrix} D_1 & 0 & 0 \\ 0 & D_2 & 0 \\ 0 & 0 & D_3 \end{bmatrix} \begin{bmatrix} 1 & L_{21} & L_{31} \\ 0 & 1 & L_{32} \\ 0 & 0 & 1 \end{bmatrix} \tag{2.7}$$

The unknown L_{ij} and D_i can be easily obtained by expressing the equalities between the upper matrix (on the left-hand-side) and its corresponding terms on the right-hand-side of Eq. (2.7). Since the LDL^T algorithm will be used later on to develop efficient, vectorized sparse algorithm, a pseudo-FORTRAN skeleton code is given in Table 2.2 (assuming the original given matrix [K] is symmetrical and full).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
1.C...... Assuming row 1 has been factorized earlier
2.        Do 11 I = 2, NEQ
3.        Do 22 K= 1, I-1
4.C...... Compute the multiplier ( Note : U represents L^T )
5.        XMULT = U(K,I) / U(K,K)
6.        Do 33 J = I, NEQ
7.        U(I,J) = U(I,J) - XMULT * U(K,J)
8.  33    CONTINUE
9.        U(K,I) = XMULT
10. 22    CONTINUE
11. 11    CONTINUE
```

Table 2.2: Skeleton FORTRAN Code For LDL ^{T}
(Assuming the matrix U is completely full)

As an example, the implementation of the LDL^T algorithm, shown in Table 2.2, for a given, simple 3×3 stiffness matrix

$$[K] = \begin{bmatrix} 2 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix} \tag{2.8}$$

will lead to the following factorized matrix

$$[U] = \begin{bmatrix} 2 & -1/2 & 0 \\ & 3/2 & -2/3 \\ & & 1/3 \end{bmatrix} \tag{2.9}$$

From Eq. (2.9), one can readily identify,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$[D] = \begin{bmatrix} 2 & & \\ & 3/2 & \\ & & 1/3 \end{bmatrix} \quad \textbf{(2.10)}$$

and

$$[L]^T = \begin{bmatrix} 1 & -1/2 & 0 \\ & 1 & -2/3 \\ & & 1 \end{bmatrix} \quad \textbf{(2.11)}$$

2.3.2 Flowchart of the Vector-Sparse LDL^T solver.

The Vector-Sparse solver developed is a collection of subroutines that implement the LDL^T Gauss elimination for matrices stored in a row-wise sparse format. In contrast to matrix that are stored in a dense, skyline or variable bandwidth fashion, sparse matrix requires special treatment before factorization. A concept of *fill-in*, the zero term that becomes non-zero after the factorization process, is introduced. Thus, minimization of fill-in terms is crucial since the amount of computation is proportional to the total number of non-zeros. The Multiple Minimum Degree, MMD, is used to minimize the fill-ins.

The implementation of a sparse Gauss elimination procedure can be broken down into several steps: the symbolic factorization (SYMFA), the numerical factorization (NUMFA1, NUMFA2, NUMFA8, for loop unrolling level 1, 2 and 8, respectively), and the forward and backward solution (FBE). An error norm check subroutine is also added to compute the absolute and relative error norm. The advantage of splitting up the computation can be seen when several linear systems have identical coefficient matrices but different right-hand sides, then only one symbolic factorization and one numerical factorization are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

needed. The different right-hand sides only require additional forward/backward operations. These strategies have also been implemented in the sparse eigensolvers in Chapter IV. Fig. 2.2 gives the flowchart of the developed vector sparse LDL^T Fortran code.

Input Data — K.* or Fort.*
nreord =1 (No / Yes)
Fill-in Minimization — MMD
Symbolic Factorization — Symfa
JU Ordering — Transa/Transa
Master(super) Dof — Supnode
Numerical Factorization — Numfa1/2/8
Forward/Backward — Fbe
Error Norm Check — Ernorn Multspa
Stop

Fig. 2.2 Flowchart of the vector-sparse LDL^T solver

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.3.3 Ordering for Gauss elimination: Symmetric matrices-MMD.

Successful implementation of a sparse equation solution algorithm depends rather heavily on the reordering method used. While the Reversed Cuthill-Mckee (RCM), or Gipspoole-Stockmyer (GS), or Gibbs-King (GK) [30, 39], reordering algorithms can be used effectively in conjunction with skyline or variable bandwidth equation solution algorithms [30], these reordering algorithms are not suitable for sparse equation solution algorithm. Ordering algorithms such as minimum-degree and nested dissection have been developed for reducing fill in factorizing sparse, symmetric matrices. Designing efficient sparse-reordering algorithms is a big task in itself, and high quality mathematical software providing efficient implementations of these algorithms is available [30]. For all the sparse codes that we have developed, the Multiple Minimum Degree (MMD) is used to reduce the fill-in.

In the case of indefinite systems, rows and columns switching are performed and the symbolic factorization cannot be completed before the numerical factorization, thus the fill-in minimization cannot be guaranteed by using MMD on the coefficient matrix. A different strategy will be suggested in Chapter III that still takes advantage of MMD.

2.3.4 Sparse symbolic factorization: SYMFA

A sparse matrix algorithm may produce new non-zeros and modify the values of the existing non-zeros of the coefficient matrix; or it may just use a given matrix without ever modifying it. The set of new non-zeros elements added to an already existing sparse matrix is refered to as fill-in terms. Memory allocations for the new fill-in terms must be available. Storage management rules, which define the internal representation of data structure, must also be enforced, identifying where and how to store each new number.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The purpose of symbolic factorization is to find the locations of all nonzero (including "fills-in" terms), off-diagonal terms of the factorized matrix [U]. Thus, one of the major goals in this phase is to predict the required computer memory for subsequent numerical factorization.

To better understand the algorithmic difficulties encountered when a sparse symmetric matrix (given in an upper triangular form) is factorized, one considers the example given in Eq.(2.5). It can be easily shown that the factorized matrix [U] will have the following form:

$$[U] = \begin{bmatrix} x & 0 & 0 & x & 0 & x \\ & x & 0 & 0 & x & 0 \\ & & x & 0 & x & 0 \\ & & & x & x & F \\ & & & & x & x \\ & & & & & x \end{bmatrix} \quad (2.12)$$

In Eq. (2.12), the symbols "x" and " F " represent the nonzero values after factorization. However, the symbol " F " also refers to "Fills-in" effect, since the original value of [K] at that location has zero entry.

For the same data shown in Eq. (2.5), if the "skyline" equation solution is adopted, [54], then the "fills-in" effect will take the following form:

$$[\bar{K}_s] = \begin{bmatrix} x & 0 & 0 & x & 0 & x \\ & x & 0 & F & x & F \\ & & x & F & x & F \\ & & & x & x & F \\ & & & & x & x \\ & & & & & x \end{bmatrix} \quad (2.13)$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

On the other hand, if the "variable-bandwidth" equation solution is adopted [55], then the "fills-in" effect (on the data shown in Eq. 2.5) will have the following form:

$$[\bar{K}_v] = \begin{bmatrix} x & F & F & x & F & x \\ & x & F & F & x & F \\ & & x & F & x & F \\ & & & x & x & F \\ & & & & x & x \\ & & & & & x \end{bmatrix} \tag{2.14}$$

Thus, for the data shown in Eq. (2.5), the "sparse" algorithm is the best (in the sense of minimizing the number of arithmetic operations, and the required storage spaces in a sequential computer environment) and the "variable-bandwidth" equation solution is the worst one. On outputs from this symbolic factorization phase, two integer arrays IU and JU will be used to store the factorized matrix.

$$IU \begin{Bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 = neq+1 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 3 \\ 4 \\ 5 \\ 7 \\ 8 \\ 8 \end{Bmatrix} \tag{2.15}$$

$$JU \begin{Bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 = NCOEF2 \end{Bmatrix} = \begin{Bmatrix} 4 \\ 6 \\ 5 \\ 5 \\ 5 \\ 6 \\ 6 \end{Bmatrix} \tag{2.16}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The following "new" definitions are used in Eqs. (2.15) and (2.16):

•NCOEF2 : The number of nonzero, off-diagonal terms of the factorized matrix [U]

•IU : Starting location of the first nonzero, off-diagonal term of the factorized matrix [U]. The dimension for this integer array is neq+1.

•JU : Column number of each nonzero, off-diagonal terms of the factorized matrix [U] (in a row-by-row fashion). The dimension for this integer array is NCOEF2. Due to "fills-in" effects, NCOEF2 >> NCOEF.

The "key" steps involved during the symbolic phase can be summarized as follows:

For each ith row of the original stiffness matrix [K]:

Step 1 :Record the locations (such as column numbers) of the original non-zero, off-diagonal terms

Step 2 :Record the locations of the "fills-in" terms due to the contributions of some (not all) appropriated, previous rows (where $1 \le j \le i-1$) Also consider if the current i^{th} row will have any immediate contribution to "future" rows.

In the symbolic factorization, the ith row of the factorized matrix is a merged list (see Section 2.2.4) of column indices of the ith row of the original matrix (step1) and column indices of fills-in due to rows 1 to i-1, that are already factorized (step2). The merge is done using a multiple switch technique (see Section 2.2.4), that results in an unordered representation structure. Eq. (2.17) summarizes the above two steps that performs the symbolic factorization of the ith row.

$$Col\# \ of \ i^{th} \ row \ of \ U = Col\# \ of \ i^{th} \ row \ of \ A + Col\# \ Fills\text{-}in \qquad (2.17)$$

where Col#: means column index.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A simple, but highly *inefficient* way to accomplish step 2 of the symbolic phase will be to identify the nonzero terms associated with the ith column. For example, there will be no "fills-in" terms on row 3 (using the data shown in Eq. 2.5), due to "no contributions" of the previous rows 1 and 2. This fact can be easily realized by observing that the associated 3rd column of [K] has no nonzero terms.

On the other hand, if one considers row 4 in the symbolic phase, then the associated 4th column will have 1 nonzero term (on row 1). Thus, only row 1 (but not rows 2 and 3) may have "fills-in" contribution to row 4. Furthermore, since $K_{1,6}$ is nonzero (=2), it immediately implies that there will be a "fills-in" terms at location $U_{4,6}$ of row 4.

A much more efficient way to accomplish step 2 of the symbolic phase is by creating two additional integer arrays ICHAINL and LOCUPDATE. ICHAINL(I=1,neq) is a circular chained list of dimension neq for the ith row. This array efficiently identifies which previous rows will have contributions to current i^{th} row. LOCUPDATE(I=1,neq) updates the starting location of the ith row during the symbolic factorization process. Besides the two additional arrays ICHAINL and LOCUPDATE, the array IU plays double roles in the actual computer implementation. At the time the ith row is being processed, the row pointers to JU corresponding to the preceding rows are stored in locations 1 to I-1 of IU. The remaining locations of IU are free. Since only column indices equal to or larger than I will be inscribed in the list JU, the locations I to neq of IU are used as the multiple switch expanded array (see Section 2.2.4) needed to perform step 2.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Considering the data shown in Eq.(2.5), the use of the above two arrays in the symbolic phase can be described by the following step-by-step procedure:

Initialize arrays : ICHAINL = {0} and LOCUPDATE = {0}

a) Consider Row i=1

Step 1 :Realizing that the original nonzero terms occur in columns 4 & 6

Step 2 :Since the chained list ICHAINL(i=1) = 0, no other previous rows will have any contributions to row 1

$$\text{ICHAINL}(4) = 1 \tag{2.18}$$

$$\text{ICHAINL}(1) = 1 \tag{2.19}$$

$$\text{LOCUPDATE}(i=1) = 1 \tag{2.20}$$

Equations (2.18-2.19) indicate that "future" row i=4 will have to refer to row 1, and row 1 will refer to itself. Eq. (2.20) states that the updated starting location for row 1 is 1.

b) Consider row i=2

Step 1 : Realizing the original nonzero term(s) only occurs in column 5

Step 2 : Since ICHAINL (i=2) = 0, no other previous rows will have any contributions to row 2.

$$\text{ICHAINL}(5) = 2 \tag{2.21}$$

$$\text{ICHAINL}(2) = 2 \tag{2.22}$$

$$\text{LOCUPDATE}(i=2) = 3 \tag{2.23}$$

Equations (2.21-2.22) indicate that "future" row i=5 will have to refer to row 2, and row 2 will refer to itself. Eq. (2.23) states that the updated starting location for row 2 is 3.

C) Consider row i=3

Step 1: The original nonzero term(s) occurs in column 5

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Step 2: Since ICHAINL(i=3) = 0, no previous rows will have any contributions to row 3.

The chained list for "future" row i=5 will have to be updated in order to include row 3 into its list:

$$ICHAINL(3) = 2 \quad \textbf{(2.24)}$$

$$ICHAINL(2) = 3 \quad \textbf{(2.25)}$$

$$LOCUPDATE(i=3) = 4 \quad \textbf{(2.26)}$$

Thus, Eqs. (2.21, 2.24, 2.25) state that "future" row I=5 will have to refer to rows 2, row 2 will refer to row 3, and row 3 will refer to row 2. Eq. (2.26) indicates that the updated starting location for row 3 is 4.

d) Consider row i=4

Step 1 : The original nonzero term(s) occurs in column 5

Step 2 : Since ICHAINL(i=4) = 1 , and ICHAINL(1) = 1 (please refer to Eqs. 2.18-2.19), it implies that row #4 will have contributions from row 1 only. The updated starting location of row 1 now will be increased by one, thus

$$LOCUPDATE(1) = LOCUPDATE(1) + 1 \quad \textbf{(2.27)}$$

Hence,

$$LOCUPDATE(1) = 1 + 1 = 2 \text{ (please refer to Eq.2.20)} \quad \textbf{(2.28)}$$

Since the updated location of nonzero term in row 1 is at location 2 (see Eq. 2.28), the column number associated with this nonzero term is column #6 (please refer to Eq. 2.5). Thus, it is obvious to see that there must be a "fills-in" term in column #6 of (current) row #4. Also, since $K_{1,6}$ = 2. (or nonzero), it implies "future" row i=6 will have to refer to row 1. Furthermore, since the first nonzero term of row 4 occurs in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

column 5, it implies that "future" row 5 will also have to refer to row 4 (in additions to refer to rows 2 & 3). The chained list for "future" row 5, therefore, has to be slightly updated (so that row 4 will be included on the list) as following

$$ICHAINL(4) = 3 \quad \textbf{(2.29)}$$

$$ICHAINL(2) = 4 \quad \textbf{(2.30)}$$

$$LOCUPDATE(i=4) = 5 \quad \textbf{(2.31)}$$

Notice that Eq. (2.30) will override Eq. (2.25). Thus, Eqs. (2.21, 2.30, 2.29) clearly show that symbolically factorizing "future" row i=5 will have to refer to rows 2, then 4 and then 3, respectively.

e) Consider row i=5

Step 1 :The original nonzero term(s) occurs in column 6

Step 2 :Since

$$ICHAINL\ (i=5) = 2 \quad (2.21,\ \text{repeated})$$

$$ICHAINL\ (2) = 4 \quad (2.30,\ \text{repeated})$$

$$ICHAINL\ (4) = 3 \quad (2.29,\ \text{repeated})$$

It implies rows #2, then 4, and then 3 "may" have contributions (or "fills-in" effects) on row 5. However, since $K_{5,6}$ is originally a nonzero term, therefore, row 2,4 and 3 will NOT have any "fills-in" effects on row 5.

f) Consider row i=6

There is no need to consider the last row i=N=6, since there will be no "fills- in" effects on the last row.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It is extremely important to emphasize that upon completion of the symbolic phase, the output array JU has to be re-arranged to make sure that the column number in each row should be in increasing order. This requirement is needed for the numerical factorization.

2.3.5 Ordered and unordered representation-TRANSA.

Sparse matrix representation do not necessarily have to be ordered, in the sense that the elements of each row can be stored in any order while still preserving the order of the rows. The symbolic factorization requires the structure IA, JA of the matrix in an unordered representation, and generates the structure IU, JU of the factorized matrix in an unordered representation. However, the numerical factorization requires IU, JU to be ordered, while IA, JA, AN can be given in an unordered representation. The algorithm that transforms a row wise representation of a matrix into a column-wise representation of the same matrix, or vice versa, has a further property that the resulting representation is ordered in the sense that the column indices of the elements in each row are obtained in the natural increasing order. Since a column-wise representation of the matrix is a row-wise representation of the transpose, the algorithm effectively transposes the matrix. Therefore, if the algorithm is used twice to transpose a matrix originally given in an unordered representation, an ordered representation of the same matrix is obtained. A symbolic transposition routine, TRANSA, that does not construct the array of non zero of the transpose structure, has been used twice to order IU, JU, after the symbolic factorization, since we are only interested in ordering JU.

2.3.6 Vectorization and finding Master (or Super) Degree-of-Freedom(dof)

There exists two approaches in performing vector computations. To illustrate these approaches, one considers the multiplication of a matrix K by a vector x.

$$\vec{y} = [K]\vec{x} \tag{2.32}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

a) Approach 1: *loop unrolling*

$$\begin{Bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{Bmatrix} = \begin{bmatrix} \vec{K}_{c1} & \vec{K}_{c2} & \dots & \vec{K}_{cn} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{Bmatrix} = \vec{K}_{c1} x_1 + \vec{K}_{c2} x_2 + \dots + \vec{K}_{cn} x_n \tag{2.33}$$

b) Approach 2: *vector unrolling*

$$\begin{Bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{Bmatrix} = \begin{bmatrix} \vec{K}_{r1} \\ \vec{K}_{r2} \\ \dots \\ \vec{K}_{rn} \end{bmatrix} \{\vec{x}\} = \begin{Bmatrix} \vec{K}_{r1} \vec{x} \\ \vec{K}_{r2} \vec{x} \\ \dots \\ \vec{K}_{rn} \vec{x} \end{Bmatrix} \tag{2.34}$$

where K_{ci} and K_{ri} (I=1,neq) are column vectors and row vectors respectively.

Loop unrolling strategy was the vectorization technique of our choice. The following pseudo Fortran coding shows the actual expansion of Eq. (2.33) for loop unrolling level 2.

The choice of the loop unrolling level depends on the machine used. For example, the optimal level for the Cray-YMP is 8 and for the IBM 3090 is 16. SUN workstations do not have vector capability. The basic requirements to apply loop unrolling is the same vector length. The nonzeros coefficients of consecutive rows must have the same column indices

We call a block of rows that satisfies the above requirements *Master (or super) degree of freedom*, or simply *supernode*. To simplify the discussion, assume that upon completion of the symbolic factorization phase, the stiffness matrix [K] has the following form:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
DO J=1,NO, LOOP ( say 2)
DO I= 1,NEQ
   Y(I)=Y(I)+X(J)* K_J (I) + X(J+1)*K_J+1(I)
ENDDO
ENDDO
c....... leftover
DO J=N0+1,NEQ
DO I=1,NEQ
   Y(I)=Y(I)+X(J)* K_J (I)
ENDDO
ENDDO
```

Table 2.3 Skeleton Fortran coding for loop unrolling

$$[K] = \begin{bmatrix} x & x & x & & & x & & & x & x & x & & x & x \\ & x & x & & & x & & & x & x & x & & x & x \\ & & x & & & x & & & x & x & x & & x & x \\ & & & x & x & & x & x & & & & & x & \\ & & & & x & & x & x & & & & & x & \\ & & & & & x & x & x & F & F & F & & x & F \\ & & & & & & x & x & x & x & F & & x & x \\ & & & & & & & x & x & x & F & & x & x \\ & & & & & & & & x & x & F & & x & x \\ & & & & & & & & & x & F & & x & x \\ & & & & & & & & & & x & x & x & x \\ & & & & & & & & & & & x & x & x \\ & & & & & & & & & & & & x & x \\ & & & & & & & & & & & & & x \end{bmatrix} \quad (2.35)$$

In Eq. (2.35), the stiffness matrix [K] has 14 dof. The symbols "x" and "F" refer to the original nonzero terms, and the nonzero terms due to "fills-in", respectively. It can be seen that rows 1-3 have same nonzero patterns (by referring to the enclosed "rectangular" region, and ignoring the fully populated "triangular" region of rows 1-3). Similarly, rows

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4-5 have same nonzero patterns. Rows 7-10 have same nonzero patterns. Finally, rows 11-14 also have same nonzero patterns. Thus, for the data shown in Eq. (2.35), the "Master" (or "Super") degree of freedom can be generated as

$$\text{MASTER} = \begin{Bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \\ 14=N \end{Bmatrix} = \begin{Bmatrix} 3 \\ 0 \\ 0 \\ 2 \\ 0 \\ 1 \\ 4 \\ 0 \\ 0 \\ 0 \\ 4 \\ 0 \\ 0 \\ 0 \end{Bmatrix} \qquad \textbf{(2.36)}$$

According to Eq. (2.36), then the "master" (or "super"") dof are dof # 1 (which is followed by 2 "slave" dof), dof # 4 (which is followed by 1 slave dof), dof # 6 (which has no slave dof!), dof # 7 (which is followed by 3 slave dof), and dof # 11 (which is followed by 3 slave dof).

In the actual Fortran code implementation, the supernode array, MASTER, is constructed by a series of If checks on consecutive rows. Different strategies can be adopted for that purpose, and the more rigid the criteria are, the less number of slaves will be obtained and vice versa. Table 2.4 gives the algorithm used to construct the array MASTER (in subroutine supnode.f).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Step 1.  Initializaton: MASTER(I)=1 for I=1,NEQ
Step 2.  To find MASTER(I)
          DO K=2,NEQ
          -Check 1: if number of nonzero of row I from column K to neq
                     is equal to number of nonzero of row K
          -Check 2: elseif column indices of row K matches those of row I
                    => same master DOF
                    else
                    => start a new master DOF I_new = K
                    endif
          ENDDO
```

Table 2.4 Algorithm for finding master DOF

In the algorithm shown in Table 2.4, for finding Master degree of freedom, the enclosed region ABD shown in Fig. 2.3 is assumed to be fully populated.

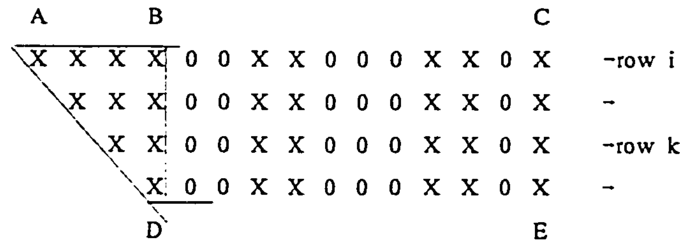


Fig. 2.3 Master Degree of freedom

2.3.7 Sparse numerical factorization with loop unrolling strategies

It is generally safe to say that sparse numerical factorization is more complicated for computer coding implementation than its skyline, or variable bandwidth cases. Main

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

difficulties are due to complex "book-keeping" (or index referring) process. In this section we assume that the symbolic factorization and ordering of the structure have been accomplished and that we have IU, JU in an ordered row-wise upper triangular format. We are now interested in the numerical part of Gauss elimination.

Let's consider the example given in Eq. (2.5). We will assume that the factorization is completed up to and including row 3, and we will examine how row 4 is processed. Row 4 has non-zeros at column numbers 4, 5 and 6. In order to find their values, we have to examine column 4, and find that the only nonzero is in the first row of this column. The nonzero elements of this first row which have column indices equal to or greater than 4 are identified. Finally,the partial factorization of the current row 4, due to the contribution from row 1 is processed.

The "key" ideas in the numerical factorization phase are still basically involved with the creation and usage of the 2 integer arrays ICHAINL and LOCUPDATE, similar to the one that has been discussed in great detail in Section 2.3.4. There are two important modifications that need to be done on the symbolic factorization, in order to do the sparse numerical factorization (to facilitate the discussion, please refer to the data shown in Eq. 2.5):
a) For symbolic factorization purpose, there is no need to have any floating points arithmetic calculations. Thus, upon completion of the symbolic process for row 4, there is practically no need to consider row 2 and/or row 3 for possible contributions to row 5. Only row 4 needs to be considered for possible contributions (or "fills-in" effects) to row 5 (since row 4, with its "fills-in", is already full). For numerical factorization purpose, however, all rows 2, then 4, and then 3 will have to be included in the numerical factorization of row 5. One

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

can see that the ICHAINL list will be more involved than the one constructed in the symbolic factorization.

b) For sparse numerical factorization, the basic skeleton FORTRAN code for LDL^T, shown in Table 2.2, can be used in conjunction with the chained list strategies (using arrays ICHAINL and LOCUPDATE). The skeleton FORTRAN code for sparse LDL^T factorization is shown in Table 2.5. Comparing Table 2.2 and Table 2.5, one immediately sees the "major differences" only occur in the second do-loop indexes, on lines 3 and 6, respectively.

```
1.c........Assuming row 1 has been factorized earlier
2.        Do 11 I = 2, NEQ
3.        Do 22 K= Only those previous rows which have contributions to
           current row I
4.c........Compute the multiplier ( Note : U represents L^T)
5.        XMULT = U(K,I) / U(K,K)
6.        Do  33 J = appropriated column numbers of row # K
7.              U(I,J) = U(I,J) - XMULT * U(K,J)
8.  33     CONTINUE
9.         U(K,I) = XMULT
10. 22     CONTINUE
11. 11     CONTINUE
```

Table 2.5: Pseudo FORTRAN Skeleton Code For Sparse LDL^T Factorization

At the begining of the numerical factorization, ICHAINL array is initialized to zero, which means that all chained lists are initially empty. To explain the numerical factorization

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

phase, let's consider Fig. 2.4 where row i is being factorized by row L. We assume that rows 1 to i-1 have already been factorized, and ICHAINL array has been consequently updated.

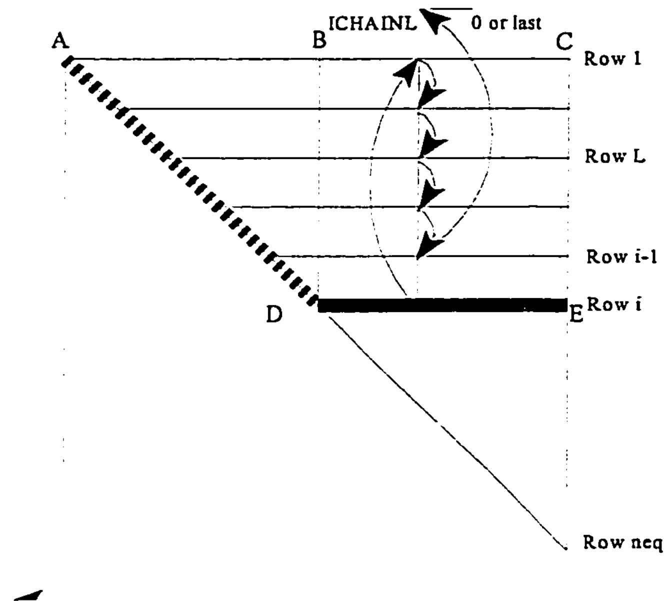


Fig. 2.4 Numerical factorization: Factorization of row i by row L

The non-zero terms of row i as well as the diagonal element of the original structure (non factorized matrix) are loaded into the multiple switch array DI (array that will contain the diagonal element of the factorized matrix on output) from location i to neq. To factorize row i, the information on the pointers to rows which have contribution to row i will be retrieved from ICHAINL array.

Let IUC = locupdate(L), IUC points to the first nonzero element of row L which has contribution to the reduction (or factorization) of row i, while IUD points to the last non zero element of row L. After the above information is collected, the multiplier is computed and the reduction of row i due to row L can be completed. It is important to note that each

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

reduced element of row i is generated and stored in an unnormalized form. It is then normilized by dividing the value by the corresponding diagonal element. Finally once the reduction for row i has been completed, the numerical values of the factorized i^{th} row are retrieved from the expanded real accumulator (see section 2.2.4) array DI and stored in the factorized matrix UN. There are two details that are very important: first of all, once the information IUC is used, the value is directly updated to point to the next non zero on row L that reduces row i, if any, as shown in Fig. 2.5. Secondly, ICHAINL is updated to include information of the "future" row that row i will update . Note also that in the symbolic factorization, row L was used once and then discarded in constructing the chain list ICHAINL, it is not the case for the numerical factorization

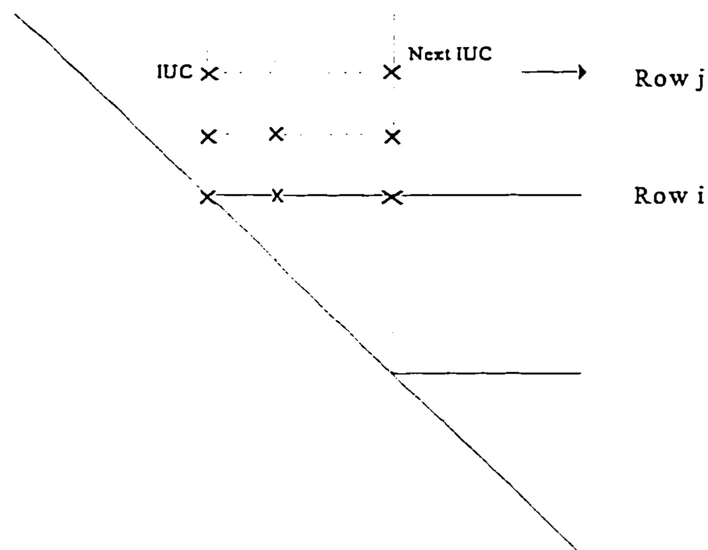


Fig. 2.5 Numerical factorization: Update location of IUC of row j

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The portion of Fortran code in Table 2.6 and Fig. 2.4 show how, in the actual Fortran implementation, the chain list ICHAINL is constructed during the numerical factorization. Two cases are considered, the first time a row is inserted in the chain list and the case of a row inserted in an existing chain list.

```
        J=JU(IUC+1)              :Column index of the next non-zero term in row L
        JJ=ICHAINL(J)            :Get information
        IF (JJ.EQ.0) GO TO 70     JJ=0 means L is the first row involved in updating J
        ICHAINL(L) = ICHAINL(JJ) :JJ≠0 Insert L in the existing chain list
        ICHAINL(JJ)=L
        GO TO 80
70      ICHAINL(J)=L              JJ=0 first time
        ICHAINL(L)=L
80      IF(L.EQ.LAST) End         L=last means no more rows that update row i
```

Table 2.6 Numerical Factorization: ICHAINL update

The vector unrolling, and loop unrolling strategies that have been successfully introduced for skyline [54] and variable bandwidth [55] equation solver, can also be effectively incorporated into the developed sparse solver in conjunction with the "master" degree of freedom strategy. Referring to the stiffness matrix data shown in Eq. (2.35), for example, and assuming the first 10 rows of [U] have already been completely factorized, our objective now is to factorize the current i^{th} row (say i= 11). By simply observing Eq.(2.35), one will immediately see that factorizing row # 11 will require the information from the previously factorized row numbers 1,2,3,6,7,8,9, and 10 (not necessarily to be in the stated increasing row numbers!) in the "conventional" sparse algorithm. Using "loop-unrolling"

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

sparse algorithm, however, the chained list array ICHAINL will point only to the "master" dof # 6, # 7 and # 1.

The skeleton FORTRAN code for LDL^T (with sparse matrix) should be modified as shown by the pseudo, skeleton FORTRAN code in Table 2.7. Comparing Table 2.5 (sparse LDL^T factorization) and Table 2.7 (sparse LDL^T factorization, with unrolling strategies), one can recognize the many similarities between the 2 sparse algorithms.

```
1.c ....... Assuming row 1 has been factorized earlier
2.          Do 11 I=2, NEQ
3.          Do 22 K=Only those previous "master" rows which have contributions to
                  current row I
4.1c .......Compute the multiplier(s) ( Note: U represents L^T)
4.2          NSLAVE DOF= MASTER (I) - I
5.1          XMULT = U(K,I) / U(K,K)
5.2          XMUL_m = U(K+m,I)/U(K+m,K+m)
5.3c ....... m=1,2 ... NSLAVE DOF
6             Do 33 J = appropriated column numbers of " master " row # K
7.1           U(I,J) = U (I,J) - XMULT * U(K,J)
7.2                        - XMUL_m *U(K+m,J)
8     33    CONTINUE
9.1           U(K,I) = XMULT
9.2           U(K+m,I) = XMUL_m
10.   22    CONTINUE
11    11    CONTINUE
```

Table 2.7 : Pseudo FORTRAN Skeleton Code For Sparse LDL^T Factorization With Unrolling Strategies

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The chained list strategies discussed earlier in Section 2.3.4 need to be modified in order not only to consider all rows that contribute to the factorization of row i, but also to include the additional information provided by the MASTER dof (refer to, for example, Eq. 2.36). The major modification that needs to be done can be accomplished by simply making sure that the chained list array ICHAINL will be pointing only toward the MASTER dof (and not toward the slave dof !). On the other hand, LOCUPDATE array is updated for the whole supernode (or master node, or master dof); thus, all rows that belong to the same supernodes will have the same IUC value.

Different levels of loop unrolling have been implemented, such as level 1 (NUMFA1), level 2 (NUMFA2) and level 8 (NUMFA8). Let's consider an example of a matrix for which the 27 rows, from row 20 to row 46, have same column numbers, or in other words, MASTER(20)=27 as shown in Fig. 2.6. Assuming that we are using loop

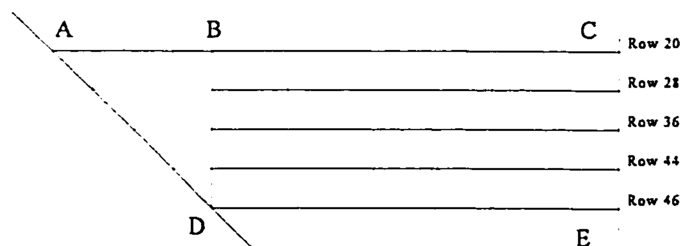


Fig. 2.6 Numerical Factorization: loop unrolling

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

unrolling level 8, the nonzeros in rectangular BCDE of Fig. 2.6 of the 27 rows will be factorized 8 rows at a time, leaving a leftover of 3 rows (rows 44 to 46), which will be factorized separately using a loop unrolling level 3. Finally the non zero terms in the triangular ABD will be factorized separately. Table 2.8 gives the order in which the vectorization has been implemented in NUMFA2 and NUMFA8.

2.3.8 Forward and backward solution

For a single right hand side vector f, the time for forward reduction and back substitution is very small as compared to the time for numerical factorization. However, for

```
          I=1
1000      continue
          II= isupnode(I)
          K=(II / LOOP)*LOOP
          Do J=I, K, LOOP (say 8)
          Factorization  with loop unrolling level 8
          ENDDO
c.........  leftover
          GO TO (10,20,30,40,50,60,70) II-K
10        unrolling level 1(do j=1,k)
20        unrolling level 2 (do j=1,k,2)
        ...
70        unrolling level 7 (do j=1,k,7)
          I = I + II    ( for next master dof)
          IF ( I.GE.NEQ ) STOP
          GO TO 1000
```

Table 2.8 Fortran Skeleton code for the vector portion of Numfa2/8

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

multiple right-hand-side vectors f, or for cases where the vector f needs to be modified repeatedly, the time for forward reduction and back substitution has to be considered more seriously.

2.3.7 Sparse matrix-vector multiplication (with unrolling strategies)

In the sparse equation solver that has been developed, after obtaining the solutions, the user has the option of computing the relative error norm. For the error norm computation, one needs to have efficient sparse matrix (with unrolling strategies) vector multiplication. Furthermore, efficient sparse matrix-vector multiplication is also required in different steps of the Subspace and Lanczos algorithms (see Chapter IV). To facilitate the discussions, let's consider the coefficient (stiffness) matrix as shown in Fig.2.7. This 14 dof matrix is symmetrical, and it has the same nonzero patterns as the one considered earlier in Eq. (2.35). The master/slave dof for this matrix has been discussed and given in Eq. (2.36). Refering to Fig 2.7, the sparse matrix-vector [A]*{x}, multiplication (with unrolling strategies) can be described by the following step by step procedure:

Step 0.1 : Perform multiplication between the given diagonal terms of [A] and vector {x}.

Step 0.2 : Consider the first "master" dof. According to Fig. 2.7, the first master dof is at row # 1, and this master dof has 2 associated slave dof. In other words, the first 3 rows of Fig. 2.7 have the same off-diagonal, nonzero patterns.

Step 1 : The first 3 rows (within a rectangular box of Fig. 2.7) of given matrix [A] operate on the given vector {x}.

Step 2 :The first 3 columns (within a rectangular box) of the given matrix [A] (shown in Fig. 2.7) operate on the given vector {x}.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Step 3 :The upper and lower triangular portions (right next to the first 3 diagonal terms of the first 3 rows of the given matrix [A] operate on the given vector {x})

Step 4 :The row number corresponding to the next "master" dof can be easily computed (using the master/slave dof information, provided by Eq. 2.36).

If the next "master" dof number exceeds N (where N = total number of dof of the given matrix [A], then stop, or else return to Step 0.2 (where the "first" master dof will be replaced by the "second" master dof etc.)

Third Step:
The upper and lower traingular region
will finally be processed

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	101.	1.	2.			3.			4.	5.	6.		7.	8.
2	1.	102.	9.			10.			11.	12.	13.		14.	15.
3	2.	9.	103.			16.			17.	18.	19.	20.	21	
4				104.	22.		23.	24.					25.	
5				22.	105.		26.	27.					28.	
6	3.	10.	16.			106.	29.	30.					31.	
7				23.	26.	29.	107.	32.	33.	34.			35.	36.
8				24.	27.	30.	32.	108.	37.	38.			39.	40.
9	4.	11.	17.				33.	37.	109.	41.			42.	43.
10	5.	12.	18.				34.	38.	41.	110.			44.	45.
11	6.	13.	19.							111.	46.	47.	48.	
12										46.	112.	49.	50.	
13	7.	14.	20.	25.	28.	31.	35.	39.	42.	44.	47.	49.	113.	51.
14	8.	15.	21.				36.	40.	43.	45.	48.	50.	51.	114.

First Step
– *These 3 rows*
– *will be processed*
– *(Dot Product Operations)*

↓ ↓ ↓

Second Step:
These 3 columns
will be processed
(SAXPY operations)

Fig. 2.7 : Sparse Matrix-Vector Multiplication With Unrolling Strategies

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.4 The Modified Oak-Ridge sparse equation solver, [25]

2.4.1 Introduction

The modified Oak-Ridge solver, which we will refer to as OakRidgeODU solver, is a collection of routines that solves a user's sparse, symmetric, positive definite linear systems via sparse Cholesky factorization (given in NASA sparse format). The user has the option of solving the matrix in its original format or to use the multiple minimun degree routine for the fill-in minimization. The original code, [25], was a set of drivers and routines that creates and solves only an artificial graph of a coefficient matrix and does not allocate and deallocate memory in an efficient manner. The modification consists of developing drivers that will read in and solve a user given matrix (in NASA format). Thus three subroutines have been developed. The first subroutine reads in the structure of the matrix in NASA format and constructs the adjacency structure. The second routine inserts the diagonal elements into the structure and creates the numerical values in the order required by the Oak-Ridge format. Memory is assigned from a single working array in the main program. No additional memory was added and all the above added routines will recycle the memory allocated during the factorization phase. The third routine is a normcheck subroutine that computes the absolute and relative error norm, making use of the sparse matrix by vector (multspa.f) multiplication.

The OakridgeODU solver has also built in the capability of making use of different sizes of the cache (in Kilobytes) on the target machine. For most machines (such as SUN Sparcstations), the optimum cache size is probably 32 or 64. For Cray type computers, the optimum cache size is 0. A study of the optimal cache size has been done using the developed solver.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2.4.2 The OakRidge data format

The original data is read in NASA format on input either as ASCII or binary. The adjancy structure (or matrix connectivity information) is constructed in subroutine Oakmain.f, and the structure of the matrix is copied and kept in another location, since the minimun degree ordering routine will destroy the original structure on exit. The structure of the non-zero coefficient in the solver, ANZ, is sparse row wise, complete and includes the diagonal values. Considering the example in Eq.(2.5), The values of ANZ are constructed row by row in the following order: The first value (in boldface) is the diagonal value followed by the lower diagonal values (underlined values) and then the upper diagonal values of each row, as shown in Eq. (2.37).

$$
\begin{array}{rlllll}
ANZ = [& \mathbf{11.} & \mathit{1.} & \mathit{2.} & & \\
 & \mathbf{44.} & \mathit{3.} & & & \\
 & \mathbf{66.} & \mathit{4.} & & & \\
 & \mathbf{88.} & \underline{1.} & \mathit{5.} & & \\
 & \mathbf{110.} & \underline{3.} & \underline{4.} & \underline{5.} & \mathit{7.} \\
 & \mathbf{112.} & \underline{2.} & \underline{7.} & &]
\end{array} \tag{2.37}
$$

2.4.3 Modification of the OakRidge solver

Fig. 2.8 gives the flowchart of the modified OakRidge solver. The solution process consists of a sequence of six distinct steps after inputting the data in NASA format: Adjancy structure, ordering, symbolic factorization, numerical factorization, Forward/Backward solution and the error norm check. The minimum degree algorithm is used to reduce the fill and work required by the factorization. An option of using the so-called "natural ordering", which is the initial ordering of the coefficient matrix, is also introduced. The symbolic factorization generated the compact data structure in which the Cholesky factor will be computed. The routine uses the efficient algorithm based on elimination trees in sparse

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

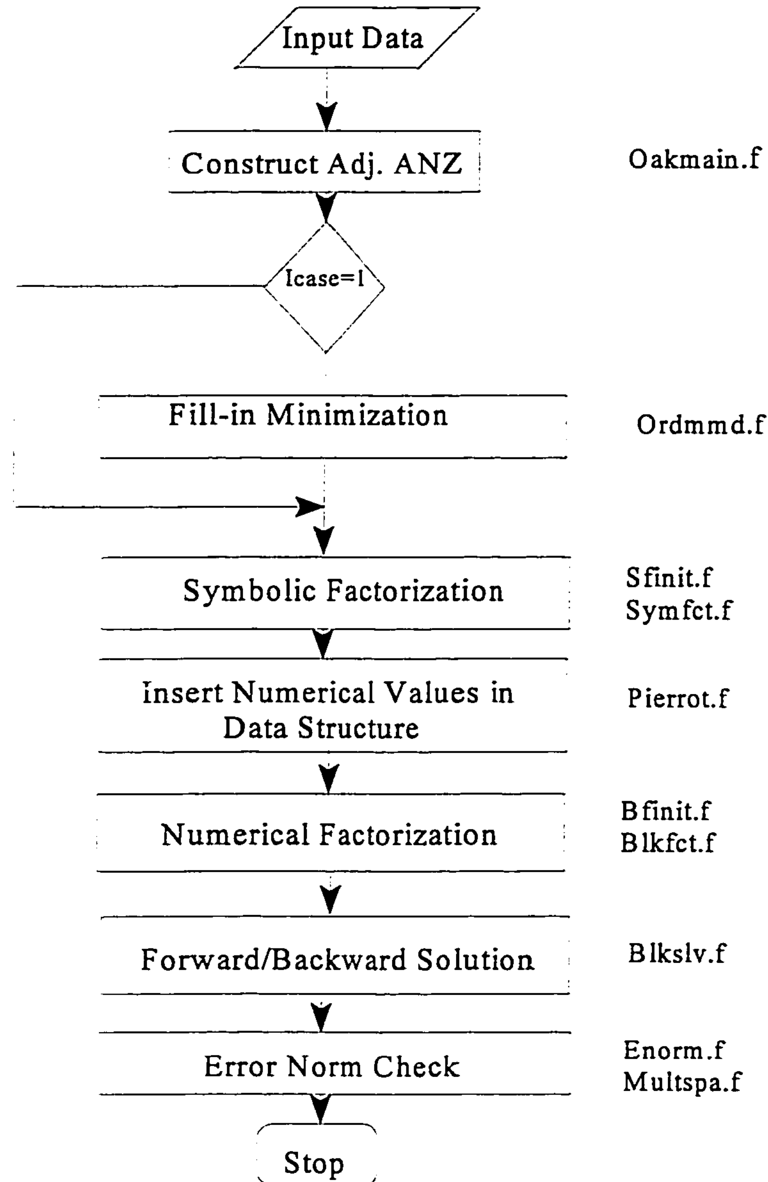


Fig. 2.8 Flowchart of the OakridgeODU Solver

factorization. The symbolic factorization is performed in two steps. The first step calls routines that implement the initialization, and the second step computes the primary symbolic factorization data structure. The numerical factorization computes the sparse

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Cholesky factor within the data structures created in the symbolic factorization phase. The left-looking block sparse Cholesky factorization algorithm has been implemented. The routine, blkfct.f, that performs the sparse block Cholesky factorization is preceded by a routine, bfinit.f, that initializes for the block factorization. The performance of this routine has been enhanced by exploiting the memory hierarchy: it operates on blocks of columns known as supernodes; it splits supernodes into sub-blocks that fit into available cache; and it unrolls the outer loop of matrix-vector products in order to make better use of available registers. The Forward/Backward phase performs the triangular solutions needed to solve the linear system.

2.4.4 Reuse of data in fast memory: CACHE

For machines with one processor, several other issues can be considered to improve the performance besides vector processing. With the continuous increase in processor speed, rapid memory access has become a very important factor in determining performance levels on several machines. To be efficient, algorithms must reuse data in fast memory (e.g., cache) as much as possible.

Let's consider a supernode that contains K columns/rows and which affect the reduction (or factorization) of J rows of the matrix. Let's define task(j,k), the modification of column/row j by a multiple of column/row k, $k<j$. One would like to consider the computation of the update (or factorize) of columns/rows J by columns/rows K during the Cholesky factorization. Suppose the operation updates q columns/rows of J with the columns/rows of K. The number of columns/rows updated may be as few as 1 or as many as $|J|$. We can compute task(J,K) as a sequence of updates task(j,K) for the q columns/rows $j \in J$. If the columns/rows of K, which happened to be stored contiguously in main memory,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

fit into cache memory, then the first task(j,K) loads the columns of K into cache, while the following q-1 tasks will have extremely fast access to this data because it is already in cache. Quite often, however, the columns/rows of a supernode do not fit into the 32K or 64 K caches used on current workstations. This can dramatically increase the number of cache misses used associated with the final q-1 tasks, as the columns/rows of K overwrite one another as they are repeatedly read into cache. To avoid this problem, the algorithm divides large supernodes into " panel" of contiguous columns/rows that fit into the cache. This simple strategy has proven effective for certain classes of problems, machines, and factorization methods used. Extremely large problems, however, may require more complicated techniques that involve both horizontal and vertical partitioning and perhaps even changes in the data structure used to store that factorized matrix.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER III

VECTOR SPARSE SOLVER FOR INDEFINITE MATRICES

3.1 Introduction

For certain classes of engineering and science applications, the symmetric coefficient matrix is not positive definite; instead, it is indefinite. Cholesky and LDL^T methods are fast and stable, and they preserve symmetry when the matrix is positive definite. However, when the matrix is indefinite, these methods can produce very inaccurate results and fail to give warning of what has occurred. It is therefore usual to recommend Gaussian elimination with partial or complete pivoting for indefinite systems, and in most cases the symmetry of the matrix is of no advantage.

Gaussian elimination with pivoting consists of switching rows and columns, operations that can be associated to a permutation matrix. There are two well known strategies for choosing permutation matrices such that Gaussian elimination will provide numerical stable solution. The first strategy, called *complete pivoting*, requires that we bring the largest element in the reduced matrix into the leading diagonal position. This strategy is called complete pivoting since we search the entire reduced matrix. The second strategy, called *partial pivoting* requires that we bring the largest element in the first column of the reduced matrix into the leading diagonal position. This strategy is called partial pivoting since we search only a part of the reduced matrix.

For positive definite systems, there is a choice of data structure. Either it may be prepared before numerical factorization starts, or it may be developed during the numerical factorization keeping pace with the stream of computed numbers. A data structure which is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

ready before initiation of numerical factorization is termed a *static* structure. Preparing it requires knowledge of the number of non-zero elements and of their positions in the matrix before they are actually factorized. The vector sparse solver for positive definite systems developed and presented in Chapter 2 uses a static structure. Static schemes present more advantages such as modularity, the symbolic and numerical steps are executed separately and consequently they can be independently optimized. Another advantage arises in the case of applications which require the repeated use of the same algorithm with different numerical values (same nozeros locations but different numerical values). Unfortunately, static data structures cannot be employed for indefinite systems. Since Gauss elimination with pivoting is used, selecting pivots using techniques such as complete pivoting, partial pivoting, or threshold pivoting amounts to permuting rows and columns, which in turn affects the location and total amount of the resulting fill-in. The consequence is that the structure of the final matrix cannot be foreseen, and decisions as to where and how to store each new fill-in non-zeros element must be made when that element has already been computed and is ready for storage. This procedure is called *dynamic* storage allocation and a dynamic structure results.

We have developed a sparse indefinite solver, the ODU-HKUST indefinite solver, with a dynamic structure,[63]. The solver uses a mixed algorithm that combines the look backward (or left looking, if lower matrix is used) and look forward (or right looking, if lower matrix is used) factorization strategies. Until the first "sick" row (a row which has nearly zero diagonal value during factorization) is encountered, the elimination is performed by looking backward and then looking forward strategies. The symbolic and numerical factorization are done simultaneously in a row after row fashion. Different pivoting

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

strategies have been developed that include those suggested by Golub, [6], and Ian Duff et al., [8]. Pivoting is performed using 1x1 or 2x2 pivoting. The use of a rotation matrix, developed by Chen Pu [63], is introduced to diagonalize the 2x2 diagonal submatrix, avoiding the difficulties of performing Gauss elimination with coupled rows. The use of 1x1 and 2x2 pivoting can be computational expensive. It involves permutations of rows/columns and may increase the fill-in of the remaining matrix. Concepts of *weighted pattern matching* of rows to be permuted and *consecutive search strategy* are introduced. In the following sections, we will explain first the pivoting strategies used and then we will describe the factorization procedure of the indefinite system adopted every time a sick row is encountered (the restarted procedure).

3.2 Symmetric indefinite systems - Pivoting strategies

3.2.1 Introduction

Although an indefinite matrix A may have LDL^T factorization, the entries in the factor could have any arbitrary magnitude:

$$\begin{bmatrix} \epsilon & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1/\epsilon & 1 \end{bmatrix} \begin{bmatrix} \epsilon & 0 \\ 0 & -1/\epsilon \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 1/\epsilon & 1 \end{bmatrix}^T \qquad \textbf{(3.1)}$$

In the above equations, some terms of [L] and [D] can be extremely (and therefore arbitrarily) large, and/or extremely small. Of course, any pivoting strategy could be invoked. However, they destroy symmetry. Symmetric pivoting, i.e., data reshuffling of the form

$$A_p \Leftarrow P A P^T \qquad \textbf{(3.2)}$$

must be used, with P as permutation matrix for this system. Unfortunately, symmetric pivoting does not always stabilize the LDL^T factorization computation. If ϵ_1 and ϵ_2 are small, then regardless of P, the matrix

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$A_p = P \begin{bmatrix} \epsilon_1 & 1 \\ 1 & \epsilon_2 \end{bmatrix} P^T \quad \textbf{(3.3)}$$

has small and large diagonal entries, and large numbers surface in the factorization. With symmetric pivoting, the pivots are always selected from the diagonal and trouble results if these numbers are small relative to what must be zeroed off the diagonal. Thus, LDL^T with symmetric pivoting can not be recommended as a reliable approach to solve symmetric indefinite systems [6]. One of the challenges is to involve the off-diagonal entries in the process while at the same time maintaining symmetry. A second challenge lies in how to take into consideration the sparsity structure of the matrix during the factorization with pivoting and in how to design an efficient Fortran code. The first challenge was solved by mathematicians in the 1970's [6,7] using either 2×2 pivoting strategies or LTL^T factorization, where T is a symmetric tri-diagonal matrix.

3.2.2 Pivoting strategies

There have been a number of pivoting strategies suggested in the literature, but most of them either destroy the symmetry structure of the matrix or fail to solve a wide range of large scale indefinite systems. New strategies are suggested and combined with the ones suggested by Golub, [6], and Duff et al, [8], for symmetric indefinite system. Let's assume that the numerical difficulties happen at the first step of the reduction (first row to be factorized). The pivoting strategies are summarized in Table 3.1. In Table 3.1, s is the order of pivoting, i.e., $s = 1$ implies diagonal pivoting and $s = 2$ implies 2×2 pivoting.

The formula to compute the parameter α (alpha) given in Table 3.1 was suggested by Golub, [6]. In our implementation we found that the value guarantees an accurate result

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$\alpha = (1+\sqrt{17})/8$ and $\lambda = |a_{1r}|$ = max off-diagonal of row 1..... (A)
if $\lambda > 0$
    if $|a_{11}| \geq \alpha\lambda$ .... (B)
    s = 1; P = I
    else
        $\sigma = a_{rp}$ = max off-diagonal of row r .... (C)
        if $\sigma |a_{11}| \geq \alpha\lambda^2$ .... (D)
            s = 1; P = I .... (E)
        else if $|a_{rr}| \geq \alpha\sigma$ ....(F)
            s = 1 and choose P so $(PAP^T)_{11} = a_{rr}$ ....(G)
        else if $|a_{pp}| \geq \alpha\sigma$ .... (H)
            s = 1 and choose P so $(PAP^T)_{11} = a_{pp}$ .... (I)
        else
            s = 2 and choose P so $(PAP^T)_{12} = a_{rp}$........... (J)
        end if
    end if
end if

Table 3.1 Pivoting strategy for symmetric indefinite system

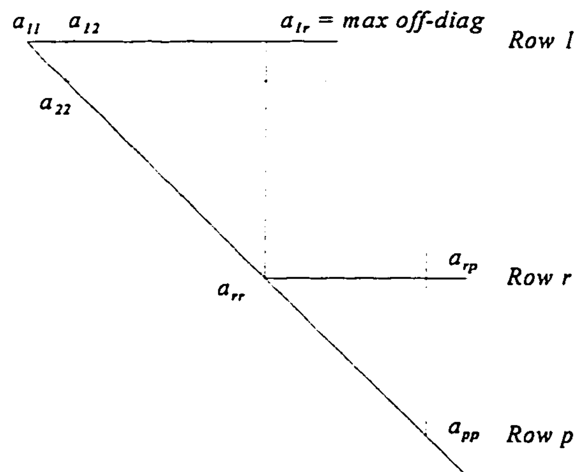


Fig 3.1 Indefinite Solver: Pivoting strategy

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

but was not the optimum in terms of performance. The value of α affects the number of 1x1 and 2x2 pivoting needed during factorization. Different values were suggested and a relaxation control parameter, *stiff*, was also added.

Before taking sparsity into consideration, let us define what is a good diagonal pivoting and what is a good 2×2 pivoting. According to Table 3.1, if $|a_{11}| \geq \alpha\lambda$ or $1 \geq \alpha\lambda/|a_{11}|$, then a_{11} is a good diagonal pivot, otherwise a_{11} is what we will call a *sick pivot*, and row 1 will be referred to as a *sick row*. The condition shown in (D) Table 3.1 can be derived as following :

From the definition of σ (see Eq. C), one has $|a_{rr}| < \sigma$. Thus $|a_{11}|\,|a_{rr}| \leq |a_{11}|\,\sigma$. From the definition of λ (see Eq.A), and from Fig. 3.1, one would like to have $|a_{11}|\,|a_{rr}| \geq \lambda^2$. Thus $\lambda^2 \leq |a_{11}|\,|a_{rr}| \leq |a_{11}|\,\sigma$ or $\lambda^2 \leq |a_{11}|\,\sigma$. Hence $\alpha\,\lambda^2 \leq |a_{11}|\,\sigma$ or $|a_{11}|\,\sigma \geq \alpha\,\lambda^2$ (since α <1, according to Eq. A). Eq.(I) of Table 3.1 indicates that row/column 1 will be exchanged with row/column p, while Eq.(J) indicates that rows/columns 1 and 2 will be exchanged with rows/columns r and p, respectively.

Similarly for a 2x2 pivoting, let us split matrix A as follows:

$$A = \begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix} \tag{3.4}$$

Following the criteria by Duff and Reid, [8], submatrix $A_{11} \in R^{2\times 2}$ is a good pivot, if $\|A_{11}\|^{-1}\Lambda \leq \begin{Bmatrix} \alpha^{-1} \\ \alpha^{-1} \end{Bmatrix}$, with $\Lambda = \begin{Bmatrix} \gamma \\ \mu \end{Bmatrix}$ as the maximum absolute row values of A^T_{21}, or in other words, this condition is equivalent to:

$$\begin{aligned} |\det A_{11}| &\geq \alpha\left(|a_{22}|\gamma + |a_{12}|\mu\right) \\ |\det A_{11}| &\geq \alpha\left(|a_{12}|\gamma + |a_{11}|\mu\right) \end{aligned} \tag{3.5}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

where $\gamma = \max_{i \neq j, j+1} |a_{ij}|$ and $\mu = \max_{i \neq j, j+1} |a_{i+1,j}|$ are the row maximum absolute value of submatrix A^T_{21} shown in Eq.(3.4) and the submatrix $A_{11} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$. Once $A_{11} \in R^{2 \times 2}$ is found to be a good 2×2 pivot, matrix A can be factorized as:

$$A = \begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} I & \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D_{11} & \\ & A_{22}^{\bullet} \end{bmatrix} \begin{bmatrix} I & L_{21}^T \\ & I \end{bmatrix} \quad (3.6)$$

with

$$D_{11} = A_{11} \quad (3.7)$$

$$L_{21} = A_{21} D_{11}^{-1} \quad (3.8)$$

$$A_{22}^{\bullet} = A_{22} - L_{21} D_{11} L_{21}^T \quad (3.9)$$

and where the partial reduced matrix $A_{22}^{\bullet}$ needs further factorization.

3.2.3 Weighted pattern matching strategy

The use of pivoting strategies usually degrades the performance. We use pivoting for the stability it induces, but despise it for the structure that it can destroy. The use of 1x1 or 2x2 pivoting in Table 3.1, once a sick row is detected, implies switching rows and columns, and consequently modifying the sparsity structure of the matrix and in most cases resulting in an increase of the number of fill-in. Therefore, pivoting should be used as a last weapon.

One of the ideas that we came up with before switching rows and columns was to compare column indices of rows to be permuted, if they match to a certain percentage (say 90 % matching): we call this *weighted pattern matching* (this idea is based upon the supernode or master node, which has already been introduced in Chapter 2). Two rows that have to be permuted, even though they are numerically stable, may introduce new fill-in after permutation. A second idea was to check the numerical stability of row sick+1 and make

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

use of it: we call this *consecutive search strategy.* Pattern matching plays an important role in minimizing fills-in. If we take into consideration the desire to keep the sparsity structure of the matrix, our criteria for a good sparse diagonal pivoting and/or 2×2 pivoting should consider the following observations:

• The non-zero off-diagonals of two rows to be interchanged, in symmetric permutation, should have similar non-zeros pattern, so that the sparsity will not change much. The similarity can be determined by the ratio between the number of column indices that match between the rows to be interchanged.

• The factorization of $L_{21} = A_{21}D^{-1}{}_{11}$, given in Eq. (3.8), introduces additional fill-ins due to the coupling of the two rows in $A^T{}_{21}$. In other words, the non-zero locations of any row in submatrix $A^T{}_{21}$ are non-zero locations of rows in $L^T{}_{21}$. So, in addition to the numerical requirement of a 2×2 pivoting, the two rows in submatrix $A^T{}_{21}$ should have similar pattern, so that less fills-in will be generated.

• The distance interval between interchanged rows plays an important role. It should be as near as possible, so that less search in the matrix will be needed. But this is not always desired. In some cases, the sick row is desired to be permuted to a row at far end, because near permutation causes sickness at neighborhood.

Suppose *lsick* is the row that is sick and will be permuted with row *lrowl1*. We call the distance interval between lsick and *lrowl1*: *jpivot* as shown in Fig.3.2. A parameter *jpivot* (distance between *lsick* and *lrowl1*) was introduced to control and limit how far we should search for a good row to switch with the sick row *lsick*. Row *lrowl1* should be numerically stable and should have almost the same pattern as *lsick*. The idea of pattern matching is important because if rows *lsick* and *lrowl1* do not have the same pattern, by

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

permuting them they may introduce more fill-in. If *jpivot* is small, meaning that row *Irowl1* is near *lsick*, the search will be small, but we will have to restart the procedure many times. On the other hand, if *jpivot* is big, the sickness appears later but we have to do a lot of search-row comparisons. Thus one can see that there is a decision to be made. In our code, *jpivot* is an input parameter.

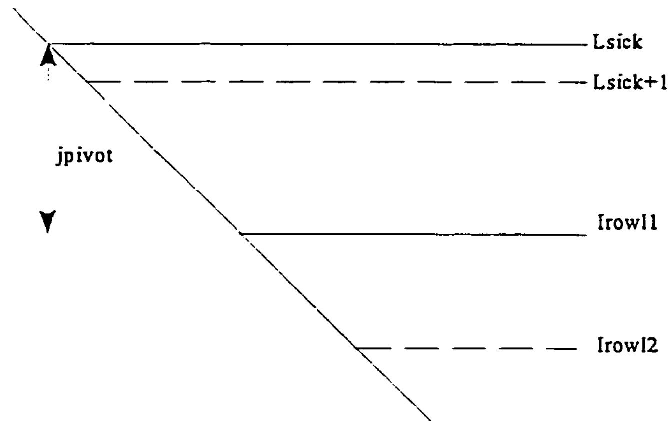


Fig. 3.2 Indefinite Solver: pattern matching

Taking into consideration the above discussions and the impact on the fill-in of the sparse matrix, Table 3.2 gives a summary of the pivoting strategies that was implemented in our indefinite solver. The row index *jb* takes into consideration the distance *jpivot*, in determing from which row to start searching for the row that will switch with the sick row.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
if a11 is a good diagonal pivot
    s = 1; P = I; exit
else
    do j = jb, neq
        If ajj is a good diagonal pivot and
        non-zero pattern of row j similar to row 1
        s = 1; choose P so (PAP^T)11 = ajj; exit
        else if submatrix [a11 a1j; aj1 ajj] is good 2*2 pivot and
        non-zero pattern of row j similar to row 1
        s = 2; choose P so (PAP^T)12 = a1j; exit
end if
end do
use Table 3.1 determine the pivot
end if
```

Table 3.2 Pivoting Strategy for sparse symmetric indefinite system with Pattern matching

3.2.4 Rotation matrix, [63]

In the look backward (or left looking) Fortran coding implementation, it is not convenient to insert a 2x2 diagonal block matrix D, although it is possible. Since the submatrix $D_{11} \in R^{2*2}$, shown in Eq. (3.7), is a non-diagonal matrix, factorizing the subsequent rows, after the 2x2 pivoting, requires special manipulations. The previous rows that contribute to the factorization of row j can be processed one row at a time, with the exception of the 2x2 block D_{11}. Thus, it is desirable to diagonalize the 2x2 block through a rotation matrix R so that the factorization can resume one row at the time, avoiding then the inconveniences of using D_{11}. Matrix D_{11} can be diagonalized as follows:

$$D_{11} = R D_{11}^{*} R^{T} \quad \textbf{(3.10)}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

with

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{3.11}$$

Thus, Eq (3.6) can be rewritten as

$$A = \begin{bmatrix} A_{11} & A_{21}^{T} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} R & \\ L_{21}R & I \end{bmatrix} \begin{bmatrix} D_{11} & \\ & A_{22}^{*} \end{bmatrix} \begin{bmatrix} R^{T} & R^{T}L_{21}^{T} \\ & I \end{bmatrix} \tag{3.12}$$

Denote $L_{21}^{*} = L_{21}R$, we can now factorize

$$A_{22}^{*} = A_{22} - L_{21}D_{11}L_{21}^{T} = A_{22} - L_{21}^{*}D_{11}^{*}L_{21}^{*T} \tag{3.13}$$

in the conventional way. In fact, by previously transforming

$$A_{21}^{*} = A_{21}R \tag{3.14}$$

the factorization for A_{22}^{*} can be processed in row-by-row fashion

3.2.5 Consecutive search strategy

The consecutive search strategy consists of checking the numerical stability of $\begin{bmatrix} a_{sick,sick} & a_{sick,sick+1} \\ a_{sick+1,sick} & a_{sick+1,sick+1} \end{bmatrix}$, if the submatrix is a suitable pivot and applies the rotation matrix. There is no interchange of rows involved. If the 2x2 pivot is not good, then one checks the stability of the diagonal value, $a_{sick+1,sick+1}$, and exchanges it with the sick row. In this case sick+1 row is a suitable pivot, and then the sickness pointer is reset to sick+1. If $a_{sick+1,sick+1}$ is not a suitable pivot, then we have to resort to the criteria in Table 3.1.

When we switch the sick row with the following sick+1 row, the value of $a_{sick,sick+1}$ does not change its location. On the other hand, after applying the rotation matrix, the pointer IUP (see definition in Chap. 2) to the first nonzero off diagonal value that reduce subsequent rows and the associate chain list will not change

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

When a sick row is detected and the consecutive search strategy can be applied, we say that we have a *recoverable sickness* and the look backward factorization can resume otherwise it is an unrecoverable sickness and the look forward factorization will procede.

3.3 Symmetric indefinite systems - Restarting

In Section 3.2, we have discussed pivoting strategies of symmetric indefinite systems, how to determine that a row is sick and suggested different strategies taking into consideration the sparsity structure of the matrix. So far, the first row is considered to be sick. In most real applications, the sickness may not occur at the beginning of the system. The code that we have developed uses a mixed look backward and look forward factorization procedure. Assume that row m+1 becomes sick in the factorization process, the first m rows will be factorized (looking backward strategy), and the procedure will restart from m+1 (looking forward strategy). Let us split matrix A accordingly as follows:

$$A = \underset{\begin{matrix} m & \quad n \end{matrix}}{\begin{bmatrix} B_{11} & B_{21}^T \\ B_{21} & B_{22} \end{bmatrix}} \begin{matrix} m \\ n \end{matrix} \tag{3.15}$$

with $B_{11} \in R^{m*m}$ and $B_{22} \in R^{n*n}$. The submatrix B_{11} can be factorized into LDL^T form

$$A = \begin{bmatrix} B_{11} & B_{21}^T \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & \\ L_{21} & I \end{bmatrix} \begin{bmatrix} D_{11} & \\ & B_{22}^{*} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ & I \end{bmatrix} \tag{3.16}$$

where

$$L_{11} D_{11} L_{11}^T = B_{11} \tag{3.17}$$

$$L_{21} D_{11}^{-1} = B_{21} \tag{3.18}$$

$$B_{22}^{*} = B_{22} - L_{21} D_{11} L_{21}^T \tag{3.19}$$

Here D_{11} is a block diagonal matrix. Its diagonal consists of 1x1 and/or 2x2 pivots.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Sickness at sick row = m+1 implies that the first row of the partially reduced submatrix B_{22}^{*} is sick. The pivoting strategies discussed in section 3.2 can be applied to that matrix. In fact the whole process will restart at sick row= m+1. The matrix B_{22}^{*} is called *partial reduced* matrix or simply partial reduction. The restarting procedure can be outlined as in Table 3.3.

```
A^(0) = A
sick = 1
do while (not the end of system)
        factorize or partially factorize A^(k)
        if (sickness detected) then
                A_new^(k) = B*_22 of A^(k)
                find pivots and permutations
                permute A_p^(k+1) = P^(k) A_new^(k) P^(k)T
        end if
end do
```

Table. 3.3: Indefinite Solver: Restarting Procedure

Until the first sick row is detected, the look backward row by row factorization (or left looking column by column factorization) is used. This corresponds to portion ABCD in Fig. 3.3, for which the elimination has been completed. The process is then restarted for portion CDE. For this portion a look forward row by row factorization (or right looking column by column factorization) is performed and the following tasks are executed.

- Simultaneous symbolic and numerical factorization
- Partial reduction
- Pivots searching
- Data management
- Permutation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

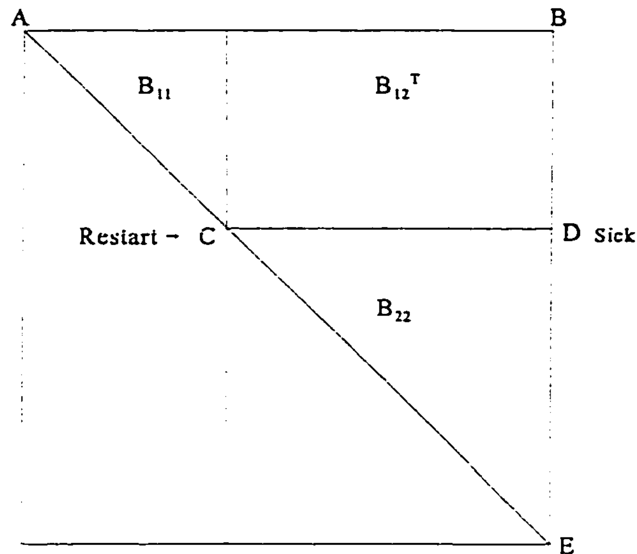


Fig. 3.3 Indefinite Solver: Restarting Procedure

3.3.1 Simultaneous symbolic and numerical factorization

When the procedure is restarted, the symbolic and numerical factorization will be carried out simultaneously. Table 3.4 gives the step-by-step procedure. Contrary to positive definite systems where static data structure can be used and the symbolic factorization can be completed on the entire matrix before the numerical factorization, in this case the symbolic factorization is executed one row at the time. Two different chain lists are used, ICHAINL and ILINK, for symbolic and numerical factorization, respectively.

```
Do j = 1, neq
    symbolic factorization
    numerical factorization
    check sickness
          :
End do
```

Table 3.4: Simultaneous symbolic and numerical factorization

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.3.2 Partial reduction

Once sickness at row sick = m+1 is detected, the subsequent rows are no longer factorized looking backward. The rows lower than m (rows m+1 to neq) will not be added in the chain lists. After the factorization of the mth row, the memory content of the matrix is as follows:

$$\begin{bmatrix} L_{mm} D_{mm} L_{mm}^{T} & U_{m,n-m} \\ SYM & A_{n-m,n-m} \end{bmatrix} \tag{3.20}$$

The factorized submatrices L_{mm}, D_{mm} and $U_{m,n-m}$ are held in the array IU,JU,UN and DU, and the part $A_{n-m,n-m}$ will be partially reduced as shown in Eq. (3.19). The result of partial reduction will be stored in the array group for U.

3.3.3 Pivot searching and Ending partial reduction criteria

We have presented different pivoting strategies and introduced the notion of weighted pattern matching and consecutive search strategy. In the actual Fortran code implementation, the search for a best pivot was done in the following order:

- If the consecutive diagonal $a_{m+2,m+2}$ value is a suitable pivot, exchange row m+1 and row m+2 immediately and move the sickness pointer to m+2; resume procedure.
- If the sick row and its consecutive row build a good 2*2 pivoting, i.e, submatrix $\begin{bmatrix} a_{m+1,m+1} & a_{m+1,m+2} \\ a_{m+2,m+1} & a_{m+2,m+2} \end{bmatrix}$ is a suitable pivot, apply the rotation matrix to uncouple rows m+1 and m+2; resume procedure.
- If a_{jj} is a good diagonal pivot (numerically stable) and non-zero pattern of row j similar to row 1 then s =1, choose P so $(PAP^T)_{11} = a_{jj}$; restart procedure. Note that when the procedure is restarted, m=1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

• If submatrix $\begin{bmatrix} a_{11} & a_{1j} \\ a_{j1} & a_{jj} \end{bmatrix}$ is good 2*2 pivot and the non-zero pattern of row j is similar to row 1, then s = 2 ,choose the permutation matrix P so $(PAP^T)_{12} = a_{1j}$; restart procedure.

• If j ≥ r and j ≥ p then use Table 3.1 to determine the pivot, where r and p are column indices of max off diagonal value of row 1 (the sick row) and r respectively (see Fig. 3.1); restart procedure.

• If matrix B_{22}^* has been formed then use Table 3.1 to determine the pivot; restart procedure.

In the partial reduction of B_{22}^*, usually not all rows are affected by the reduction. Let's call *jend* the last row to be affected by the partial reduction. This means rows from *jend+1* to neq will not be affected by the reduction. The row *jend*, can be located before the completion of $B_{22}^* = B_{22} - L_{21}D_{11}L_{21}^T$ calculation, i.e, the partial reduction can be ended in advance, if the permutations in all those cases affect the rows between m+1 and jend. Fig. 3.4 shows the partial reduced zone and the location of *jend.*

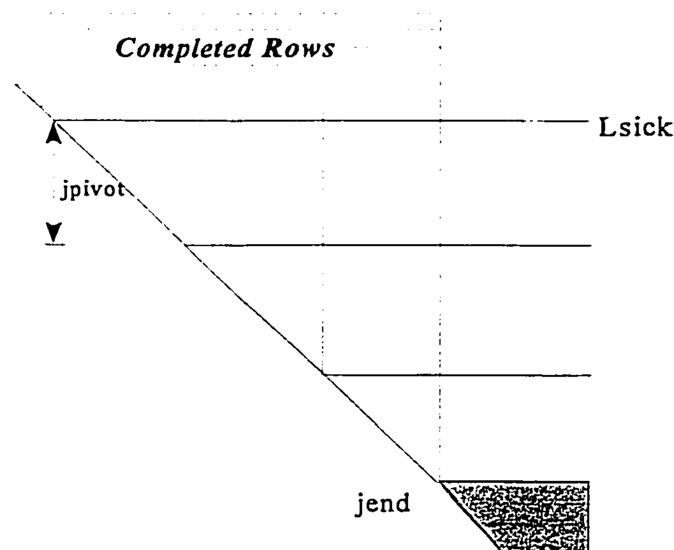


Fig. 3.4 Indefinite Solver: Ending Partial reduction zone

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

3.3.4 Data management

We use a large integer array IWORK(1: *mtot*) as a working space where mtot is the maximum computer memory available. The value of *mtot* is machine dependent and is an input control parameter. All array are allocated from this array. The known, fixed dimensions for arrays IA(1:neq+1), AD(1:neq), IU(1:neq+1), UD(1:neq), ILINK(1:neq), ICHAINL(1:neq) etc. are placed at the beginning of IWORK(1:mtot). The remaining memory will be divided into 2 segments, where the first segment holds UN and AN, and is twice as big as the second segment which holds JU and JA (because *real*8* and *integer*4* declarations are used in the coding). Arrays AN(1:ncoef) and JA(1:ncoef) are placed at the bottom of the first and second segment, respectively. It should be noted that the dimension of AN and JA changes every time the procedure restarts. Fig. 3.5 shows the suggested memory allocations.

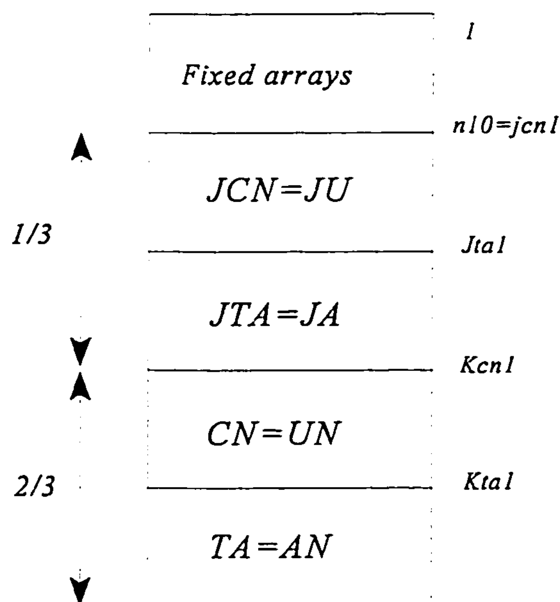


Fig. 3.5 Indefinite Solver: Memory allocation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In order to keep the consistency of the program and to take into consideration the memory allocations and data movement during the restarting procedure, the restarting procedure algorithm can be rewritten as in Table 3.5.

```
Last sick = sick row
do while ( not the end of system)
    • restart LDL^T at the last sick row,
    • perform partial factorization
    • if sickness is detected, then
        -find suitable pivot and permutations
        -permute JU,UN to JA, AN
        -rearrange IA, JA,AN and AD
    endif
end do
```

Table 3.5 Restart algorithm of symmetric indefinite solver

The partially reduced submatrix B_{22}^{*} is the matrix that is only considered when the procedure is restarted. Thus during the factorization, B_{22}^{*} is placed in the array JU and UN while permuting rows, $PB_{22}^{*}P^{T}$. The pointer to rows of JU and UN, from row *sick* to row *jend*, will constitute the beginning row of the new restarted array JA and AN, respectively. Because of the similarity of structure betwen JU and UN and between JA and AN, to simplify the discussion, we will only consider the memory management of JU and JA and explain the data movement between the two arrays once the procedure is restarted. Let's

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

consider the genaral case where the process restarts at row m. We will distinguish three cases for the memory allocation before restarting in Fig. 3.6.

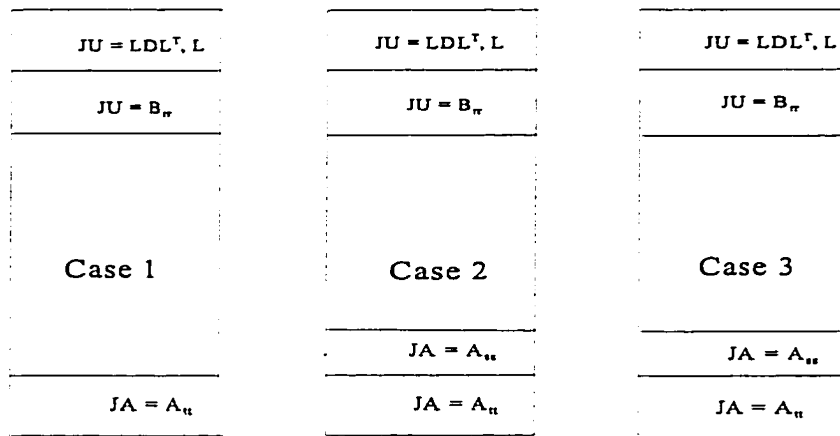


Fig. 3.6: Indefinite Solver: Memory Reallocation

Case1:

$$\begin{bmatrix} L_{mm} D_{mm} L_{mm}^T & L_{rm}^T & 0 \\ & B_{rr}^* & B_{rt}^* \\ SYM & & A_{tt} \end{bmatrix} \tag{3.21}$$

In Fig.3.6 we assume that the first m rows have been completely factorized, the partial reduction of the subsequent rows has been completed, and the symmetric permutations determined by Table 3.1 affect only the rows in the middle part of Eq. (3.21), which corresponds to submatrices B_{rr} and B^*_{rt}. In this case the submatrix A_{tt} remains unchanged. While doing permutations, the part $JU = B_{rr}$ moves to the top of $JA = A_{tt}$.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Case 2:

$$\begin{bmatrix} L_{mm}D_{mm}L_{mm}^T & L_{rm}^T & 0 & 0 \\ & B_{rr}^{*} & B_{rs}^{*} & B_{rt}^{*} \\ & & A_{ss} & A_{st} \\ SYM & & & A_{tt} \end{bmatrix} \quad (3.22)$$

In this second case, the symmetric permutations determined by Table 3.1 affects only the rows in the two middle parts of Eq. (3.22), which correspond to submatrices B_{rr}^{*}, B_{rs}^{*}, B_{rt}^{*}, A_{ss} and A_{st}. The elements of A_{tt} remain unchanged after the permutations. The memory reallocation is divided into two steps. In the first step, the part JA = A_{ss} moves to the bottom of JU = B_{rr}, and then in the second step, two parts JU=B_{rr} and JA = A_{ss} are reallocated to the top of JA = A_{tt}.

Case 3:

$$\begin{bmatrix} L_{mm}D_{mm}L_{mm}^T & L_{rm}^T & U_{ms} & 0 \\ & B_{rr}^{*} & B_{rs}^{*} & B_{rt}^{*} \\ & & A_{ss} & A_{st} \\ SYM & & & A_{tt} \end{bmatrix} \quad (3.23)$$

In this third case, the symmetric permutations determined by the searching in the do-loop for j of Table 3.2 affects only the rows in the second part of Eq.(3.23), which corresponds to submatrices B^{*}_{rr}, B^{*}_{rs} and B^{*}_{rt}. The elements of A_{ss}, A_{st} and A_{tt} remain unchanged, and the permutations of rows are completed. The portion of JU = B_{rr} moves to the top of array JA = A_{ss}. It should be noted that the factorization after restarting still needs U_{ms}; therefore, in both the symbolic and numerical factorization of the submatrix A_{ss}, the chain lists ICHAINL and ILINK should point to rows in submatrix U_{ms}.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In each of the above cases, the submatrices from B^*_{rr} to A_{tt} constitute the new matrix A to be considered when the procedure restarts. The matrix is stored in sparse format as a group of arrays IA, JA, AN, AD. The new group of arrays holds the data from the last sick = m+1 row to the end of the matrix. In all cases, symmetric permutations affect only the parts of new matrix A. It must be pointed out that the permutations do not affect the portion of the matrix already factorized, L_{mm}, D_{mm}, $L_{mm}{}^T$, $L_{rm}{}^T$ and U_{ms}.

3.3.5 Permutation

The stabilization of Gaussian elimination that is developed involve data movements associated with switching rows and columns. If a square matrix P of order n is a permutation matrix , and p(1:n) is the desired permutation of n rows of a matrix; one can definite P as:

$$P_{i,p_i} = 1 \quad \text{and} \quad P_{ij} = 0 \quad \text{otherwise}$$

Every row and every column of P contains just one element equal to 1, the remaining elements of the row (or column) are equal to 0 and P is orthogonal ($P^T = P^{-1}$). If a matrix A is premultiplied by P, the original row p_i of A will become row i of the resulting matrix PA. P can be stored in the computer memory as a vector of integers: the integer at position i is the column index of the unit element of row i of P. Indeed, by knowing the permutation, a vector $X \in R^n$ can be overwritten as follows:

```
for i=1:n
        X(i) ⇔ X(p(i))
end
```

Here, the " ⇔ " notation means "swap contents".

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It should be noted that no floating point arithmetic computation is involved in a permutation operation. However, permutation matrix operations often involve the irregular movement of data and therefore can represent a significant computational overhead.

Traditionally, column indices in JA, are considered in the ascending order at a particular row. In our case, we found that it was not necessary to arrange elements of JA in ascending order; in other words , JA can be unordered. With minor changes to the subroutine to perform the transposition of a matrix, one can write a subroutine to perform the permutation of rows and column of a matrix A. However, one will have to apply the subroutine on the structure of the entire matrix. A different subroutine was specialy designed from scratch to consider only a portion of a matrix and to perform only the permutation of a few rows, cutting down the overhead cost associated in considering the entire matrix.

3.4 Forward reduction and Back substitution

Due to the restarting scheme, the permutations affect the only matrix part B^*_{22}, so we can not claim the final results after factorization as:

$$P^{(P)}P^{(P-1)} \ldots P^{(1)}A(P^{(1)})^T. \ldots (P^{(P-1)})^T(P^{(P)})^T = LDL^T \qquad \textbf{(3.24)}$$

Eventhough the permutations vectors are known, they are applied on the reduced submatrix B^*_{22} when the procedure is restarted and not on the original matrix. The step by step procedure in Table 3.6 shows the implication of the permutation and rotation matrices on the load (or right-hand-side) vector and how one can recover the solution during the forward and backward substitution. In practice, forward and back substitution only require very little time as compared to factorization.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
do j = 1, neq                              ! Forward reduction
if j is an index of sick row:      f ⇐ P^(k)f
if j is an index of rotation row: f ⇐ R^(l)f
do i = indices in JA for row j
        y_i= f_i - L_ij *y_j
end do
        y_i= y_i / D_ii
enddo
do j= neq, 1,-1                  ! Back substitution
        if j is an index of rotation row:  y ⇐ (R^(l))^T y
        if j is an index of sick row:     y ⇐ (P^(k))^T y
        do i = indices in JA for row j
                x_i = y_i - L_ij * x_i
        end do
end do
```

Table 3.6 : Forward Reduction and back substitution

3.5 Reordering of indefinite systems

As mentioned in the introduction of Section 3.1, a static structure cannot be implemented for an indefinite solver that uses pivoting strategies. Since rows and columns are permuted during the factorization process, a fill-in minimization cannot be performed a priori as it was the case for positive definite systems and the structure of the final matrix after factorization cannot be forseen.

The idea that we came up with was to try to maximize the portion on which the look backward factorization is performed and to deal with unstable rows at the end of the matrix. The Multiple Minimum Degree (MMD) was performed on the entire matrix and the rows/columns corresponding to the zero diagonal have been pushed to the end of the matrix as shown in Fig. 3.7. By using this strategy, there has been improvement in the performance, but one cannot guarantee that the fill-in minimization during the pivoting was

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

optimum. A better strategy would have been to first push all the zeros at the end of the matrix (B-F) and then perform MMD only on a portion of the matrix, say ADBC, at the same time minimize the fill-in of the coupling block CEFB. If possible, we prefer to reorder the matrix such that all the non-zero coefficients of CEFB reside in the lower portion of the coupling block (HBIG).

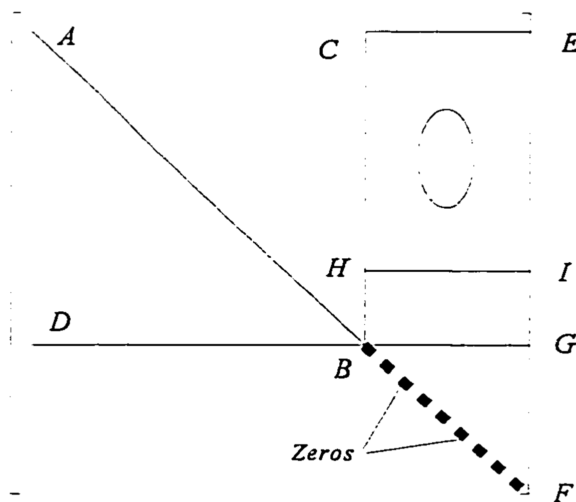


Fig. 3.7 Indefinite Solver : Fill-in Minimization

3.6 The modified MA27 sparse indefinite solver.[8,66]

3.6.1 Introduction

The MA27 is a software package from the Harwell subroutine library developed by Duff et al., [8], that uses a sparse variant of Gaussian elimination to solve a sparse indefinite system of linear equation. The MA27 uses the multifrontal approach and contains three majors subroutines. The MA27A/AD accepts the pattern of the matrix and chooses pivots for Gaussian elimination using a selection criterion to preserve sparsity. The subsidiary

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

information for actual factorization are constructed by the subroutine MA27/BD. The pivots are chosen from the diagonal using the minimun degree criterion and employing a generalized element model of the elimination. The elimination is represented as an assembly and elimination tree with the order of elimination determined by the depth-first search of the three.

The MA27B/BD factorizes the matrix by using the assembly and elimination ordering generated by MA27/AD. At each stage in the multifrontal approach pivoting and elimination are performed on full submatrices and, when diagonal 1x1 pivots would be numerically unstable, 2x2 pivots diagonal blocks are used. The actual pivot sequence used may differ slightly from that of MA27A/AD if the matrix is not definite.

The MA27C/CD uses the factors generated by MA27B/BD to solve a system of equation Ax=b. Since the information passed from one subroutine to the next is not corrupted by the second, several calls to MA27B/BD for matrices with the same sparsity pattern but different values may follow a single call to MA27A/AD, and similarily MA27C/CD can be used repeatedly to solve for different right-hand-side vectors b.

3.6.2 MA27 data format and control parameters

The data format used in the MA27 differs from the NASA format. The matix is represented by 3 arrays, IRN , ICN and A. The one dimensional real array A contains the diagonal values as well as the off diagonal values and will be of dimension ncoef+neq. The integers arrays IRN and ICN contain the row and column indice of each value in A respectively and has same dimension as A.

The following control parameters are used:

N : integer variable set by the user to the order neq of the matrix

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

NZ : number of non zeros entries in the matrix (nz=ncoef+neq)

LA : integer variable which must be set by the user to the length of A. It is advisable to allow a slightly greater value because the use of numerical pivoting might increase the storage requirements marginally.

NRLNEC and NIRNEC are integer variables. On exit from MA27AD/AD they give the amount of REAL and integer words required respectively for successful completion of the factorization, provided no numerical pivoting is performed. Numerical pivoting may cause a higher value to be required.

IKEEP: integer array of length equal to 3*neq. It is used if the user wishes to input the pivot sequence.

IFLAG: is an integer variable which must be set to zero if a suitable pivot order is to be chosen automatically, or to 1 if the pivot order set in IKEEP is to be used. On exit from MA27/AD, a value of zero indicates that the subroutine has performed successfully. A nonzero values means that an error has been detected.

3.6.3 Modified MA27 solver: ODUMA27

MA27 failed to solve our benchmark indefinite test problems. We acknowledge here the constructive discussions with J. Qin, [20, 66], to implement a new pivoting criteria to the existing sequence in order to solver these problems. The modified MA27 sparse solver appears to be fast and reliable. The modification consisted not only of stiffening the pivoting strategies (by reducing the number of required 2x2 pivoting during factorization, whenever possible, for saving computational time, see Section 3.3), but also of adding the capability of reading data in NASA row-wise sparse format.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER IV

SPARSE SUSPACE AND LANCZOS ITERATION FOR THE SOLUTION OF POSITIVE DEFINITE AND INDEFINITE SYSTEMS.

4.1 Introduction

The generalized eigen-equations, in matrix notation, can be expressed as

$$[K]\ [\phi] = \lambda\ [M]\ [\phi] \quad \textbf{(4.1)}$$

In Eq. (4.1), matrices [K] and [M] represent the structural stiffness and mass, respectively. Matrices [λ] and [ϕ] represent the eigenvalues and eigenvectors, respectively. The dimension (or degree-of-freedom) of matrices in Eq. (4.1) is N.

Much attention has been directed toward effective algorithms for the calculation of the required eigensystem in the problem of Eq. (4.1). Because the "exact" solution of the required eigenvalues and corresponding eigenvectors can be expensive when the order of the system is large, approximate solution techniques have been developed. The approximate solution techniques have primarily been developed to calculate the lowest few eigenvalues and corresponding eigenvectors in the problem of Eq. (4.1), when the order of the system is large. However, the problem of calculating the few lowest eigenpairs of relatively large-order systems is very important and is encountered in all branches of engineering.

Vector sparse Subspace and Lanczos iteration eigensolvers have been developed for positive definite and indefinite systems. Besides the use of sparse technology in all the algebraic manipulation and data structure involved, the developed solvers in Chapter II and III have been incorporated in the Fortran code implementation for efficient eigen-solution.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.2 Subspace Iteration, [1,40-43]

4.2.1 Basic Subspace Iteration Algorithm

The Subspace iteration method developed by K.J. Bathe, [1], consists of establishing q starting iteration vectors, q>p, where p is the number of eigenvalues and eigenvectors to be calculated. It extracts the "best" eigenvalue and eigenvector approximations from the q iteration vectors, by using inverse iteration on the q vectors and Ritz analysis.

The basic objective of the Subspace iteration method is to solve for the smallest p eigenvalues and corresponding eigenvectors satisfying Eq. (4.1). In addition to the relation in Eq. (4.1), the eigenvectors also satisfy the orthogonality conditions

$$\Phi^T K \Phi = \Lambda \ ; \quad \Phi^T M \Phi = I \tag{4.2}$$

Where I is a unit matrix of order p because ϕ stores only p eigenvectors. The essential idea of the Subspace iteration method uses the fact that the eigenvectors in Eq. (4.1) form an M-orthonormal basis of the p-dimensional least dominant subspace of the matrices K and M, which we will call E_∞. The starting iteration vector span E_1, and iteration continues until, to sufficient accuracy, E_∞ is spanned. Thus, the total number of iterations depend on how "close" E_1 is to E_∞ and not on how close each iteration vector is to an eigenvector. Hence, the effectiveness of the algorithm lies in that it is much easier to establish a p-dimensional starting subspace that is close to E_∞ than to find p vectors that are each close to a required eigenvector. The selection of starting iteration vectors is a very important part of the iteration procedure.

The first step in the Subspace iteration method is the selection of the starting iteration vectors X_1. The choice of the starting iteration vectors is important in the sense that it can reduce the number of iteration needed for convergence; for example, if the starting vectors

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

span the least dominant Subspace, the iteration converges in one step. In this section we describe the starting vectors that have been used in our code.

Let $[X]_{n\times q}$ be the matrix that contains the starting iteration vector

$$[X] = [\vec{x}_1, \vec{x}_2, \vec{x}_3, \ldots, \vec{x}_{q-1}, \vec{x}_q] \tag{4.3}$$

where $\vec{x}_i$ are q vectors of dimension $n\times 1$. The step by step algorithm to construct the starting iteration vectors can be summarized as follows:

Step1: $\vec{x}_1(i) = dmass(i) \qquad i=1,n$

Step2:. $\vec{x}_i = \vec{e}_i \qquad for \qquad i=2,q-1$

Where $\vec{e}_i$ are unit vectors with entries +1 at the degree of freedom with smallest ratios

$$w_i = \frac{k_{ii}}{m_{ii}} \qquad \text{with} \quad m_{ii} = dmass(i) \tag{4.4}$$

Step3: $\vec{x}_q = random \quad vector$

Table 4.1 Step by step algorithm for starting iteration vector

An important procedure that is used extensively in the solution of eigenvalues and eigenvectors is shifting. The purpose of shifting is to accelerate the calculations of the required eigensystem. In the solution of Eq. (4.1), we perform a shift ρ on K by calculating

$$\hat{K} = K - \rho M \tag{4.5}$$

and we then consider the eigenproblem

$$\hat{K}\psi = \mu M \psi \tag{4.6}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

To identify how the eigenvalues and eigenvectors of Eq.(4.1) are related to those of Eq. (4.6), using Eq.(4.5), we rewrite (4.6) as follows

$$K\psi = \gamma M \psi \tag{4.7}$$

where $\gamma = \rho + \mu$. However, Eq.(4.7) is in fact, the eigenproblem $K\Phi = \lambda M\Phi$, and since the solution of this problem is unique, we have

$$\lambda_i = \rho + \mu_i \qquad \text{and} \qquad \phi_i = \psi_i \tag{4.8}$$

In other words the eigenvectors of $\hat{K}\psi = \mu M \psi$ are the same as the eigenvectors of $K\phi = \lambda M \phi$, but the eigenvalues have been decreased by ρ.

4.2.2 Subspace Iteration step by step Algorithm

Subspace iteration algorithm can be used effectively to obtain the lowest p eigen-pair solutions. The algorithm can be conveniently described by the following step-by-step procedures shown in Table 4.2.

4.2.3 Subspace Iteration for positive definite systems: LDL^T

The step by step algorithm in Table 4.2 has been coded for the solution of positive definite systems. The starting iteration vector in step 1 has been constructed following the algorithm in Table 4.1. The system of equation that results in Eq. (4.9) has been solved using the developed vector sparse solver for positive definite system in Chapter II. Matrix K is factorized only once and the forward and backward solution is called q times for the multiple right hand side $[Y_k]_{Nxq}$. Once the reduced stiffness matrix and mass matrix have been constructed, following Eq.(4.11) and Eq.(4.13) respectively, the reduced eigen-problem is solved using Jacobi for all q eigenvalues and eigenvectors and ordered in ascending order. The process is then repeated until the convergence is achieved. All the matrix manipulations involved are performed using sparse technology.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Step 1: Select the starting iteration vectors $[Y_1]_{Nxq}$ where $q << N$

Step 2: Factorize the structural stiffness matrix

$$[K] = [L][D][L]^T \tag{4.9}$$

In Eq. (2), [L] is the lower triangular matrix, and [D] is the diagonal matrix

Step 3: For k = 1,2, ... Maxiter, where Maxiter represents the input maximum number of iterations, the following tasks need to be done

Step 4: Solve $[\Phi_{k+1}]_{Nxq}$ from the following matrix equations

$$[K][\Phi_{k+1}]_{N\times q} = [Y_k]_{N\times q} \tag{4.10}$$

Step 5: Compute the reduced stiffness matrix

$$[K^R_{k+1}]_{q\times q} = [\Phi_{k+1}]^T_{q\times N}[Y_k]_{N\times q} \tag{4.11}$$

Step 6: Compute the reduced mass matrix

$$[\bar{Y}_{k+1}]_{N\times q} = [M]_{N\times N}[\Phi_{k+1}]_{N\times q} \tag{4.12}$$

$$[M^R_{k+1}]_{q\times q} = [\Phi]^T_{q\times N}\,[\bar{Y}_{k+1}]_{N\times q} \tag{4.13}$$

Step 7: Solve the reduced eigen-equations

$$[K^R_{k+1}]_{q\times q}[Q_{k+1}]_{q\times q} = [M^R_{k+1}]_{q\times q}[Q_{k+1}]_{q\times q}[\Omega^2_{k+1}]_{q\times q} \tag{4.14}$$

The eigenvalues $[\Omega^2_{k+1}]$ and the associated eigenvectors $[Q_{k+1}]$ need to be arranged in the ascending orders (for example $\Omega^2_1 < \Omega^2_2 < \Omega^2_3 < \ldots$)

Step 8: Find an improved approximation to the eigenvectors

$$[Y_{k+1}]_{N\times q} = [\bar{Y}_{k+1}]_{N\times q}[Q_{k+1}]_{q\times q} \tag{4.15}$$

Step 9: Check for convergence. The iterative process will be stopped if either convergence is achieved, or the maximum number of iteration (= Maxiter) is reached (or else, return back to step3).

Table 4.2: Step-by step Basic Subspace Algorithm

The error bounds and check for convergence of eigenvalues are performed at the end of each iteration. Assuming that in iteration (k-1) the eigenvalue approximation $\lambda_i^{(k)}$, i=1,..,p, have been calculated. Then the convergence tolerance is computed, [1], in the form

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$\left[1 - \frac{(\lambda_i^{(k)})^2}{(\phi_i^{(k)})^T(\phi_i^{(k)})}\right]^{1/2} \leq tol: \qquad i=1,..,p \tag{4.16}$$

where $\phi_i^{(k)}$ is the eigenvector corresponding to the eigenvalue $\lambda_i^{(k)}$ and tol $=10^{-2s}$ when the eigenvalue shall be accurate to about 2s digits. For example, if we iterate until all p bounds in Eq. (4.16) are smaller than 10^{-6}, we find that λ_p has been approximated to at least six digit accuracy, and the smaller eigenvalues have usually been evaluated more accurately.

Once the error bounds and the convergence on the eigenvalues have been checked, the "true"error norm check is computed as follows:

$$\frac{\| K\phi - \lambda M\phi\|_2}{\|K\phi\|_2} \leq Toler2 \tag{4.17}$$

Our efficient sparse matrix times vector multiplication is used in evaluating Eq. (4.17)

4.2.4 Subspace Iteration for Indefinite systems: ODU-HKUST, ODU-MA27

The step by step algorithm of Table 4.2 has been implemented for the solution of indefinite systems. The starting iteration vector (see Table 4.1) has been modified from Eq.(4.4). The value of w(i) is set to zero when the ratio m_{ii}/K_{ii} is infinity (or undetermined). Two solvers for the solution of indefinite systems, the ODU-HKUST solver and the ODU-MA27 solver, have been developed in Chapter III. These solvers have been incorporated in the Subspace iteration algorithm in factorizing the matrix K of Eq.(4.9). An input control parameter is provided to choose the type of solver. The error bound and convergence check are performed as shown in Eq. (4.16). The "true" error norm is also computed according to Eq.(4.17).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

4.3 Lanczos Iteration [1,3,44]

4.3.1 The Lanczos Iteration Algorithm

The Lanczos algorithm for the solution of generalized eigenvalue problems has been receiving a lot of attention due to its computational efficiency . The Lanczos method was originally developed to evaluate eigensolution of matrices through a Rayleigh-Ritz reduction of the eigensystem to a tridiagonal form. The eigenvectors are constructed by forming a linear combination of a set of vectors, known as Lanczos vectors, computed during the course of the Lanczos algorithm. Intensive research in past years has resolved a number of difficulties concerning the stability of the Lanczos process. It is now widely accepted as the method of choice for determining a few eigenpairs of large sparse problems.

Let's consider the following generalized eigenvalue problem:

$$K \phi = \omega^2 M \phi \tag{4.18}$$

or

$$K_\sigma \phi = \omega_\sigma^2 M \phi \tag{4.19}$$

where K and M are structural stiffness matrix and mass matrix, respectively, K_σ= K-σM, σ is the shift value and $\omega_\sigma^2 = \omega^2 - \sigma$. Instead of solving Eq. (4.18), or Eq. (4.19) directly, the Lanczos algorithm generates a tri-diagonal matrix T_m

$$T_m = \begin{bmatrix} \alpha_1 & \beta_2 & & & \\ \beta_2 & \alpha_2 & \beta_3 & & \\ & \beta_3 & \alpha_3 & & \\ & & & \ddots & \\ & & & & \beta_m \\ & & & \beta_m & \alpha_m \end{bmatrix} \tag{4.20}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

through the following three-term recurrence formula:

$$r_j = \beta_{j-1} q_{j-1} = K_\sigma^{-1} M q_j - \alpha_j q_j - \beta_j q_{j+1} \tag{4.21}$$

or in matrix form:

$$[K_\sigma^{-1} M] Q_m - Q_m T_m = \{0.0....r_m\} = r_m e^T_m \tag{4.22}$$

$$T_m z = \theta z \tag{4.23}$$

where $e^T_m = (0,0,...1)$, Q_m is a N×m orthogonal matrix with columns $q_j = 1,2,3$ ...m, and m is usually much smaller than N. By solving the following reduced eigensystem, the eigensolution of Eq. (4.19) can be obtained as

$$\omega_\sigma^2 = \frac{1}{\theta} \tag{4.24}$$

$$\phi = Q_m z \tag{4.25}$$

For most structural engineering problems, only a few lowest frequencies and the corresponding mode shapes are required, so we have m << N, which leads to a significant savings in the number of operations.

A partial restoring orthogonality scheme and a convergence criterion are developed and incorporated into the basic Lanczos algorithm, which is described in a step-by-step procedure, shown in Table 4.3.

Various reorthogonalization schemes have been developed to increase the efficiency of Lanczos algorithms [44-48]. However, for very large problems where factorization, forward/backward substitution and matrix-vector multiplication are the major operations, the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

cost of reorthogonalization becomes less important than for small problems, since only a few lowest eigenpairs are desired. In this work, a simple way of reorthogonalization is adopted.

First, for any new Lanczos vector q_j, one calculates

$$E_i = q^t_i M q_j \qquad (i = 1,2,....,j-1) \tag{4.26}$$

If $E_i > E$, then q_j should be orthogonal to q_i with respect to M, where E is a parameter related to the machine parameter E_o such that $1+E_o > 1$. Usually, E is taken as:

$$E = \sqrt{E_o} \tag{4.27}$$

Eq. (4.27) is called semi-orthogonality [46] condition.

One major advantage of the Lanczos algorithms lies in their ability to terminate the iteration process as soon as the required eigenpairs have converged. In this work, the following error bound for eigenvalues is used (after solving Eq. 4.23 in step 12)

$$\text{ERROR } (i) = \left|\frac{\lambda_k - \theta_i}{\theta_i}\right| = \left|\beta_{j+1}\frac{Z^{(i)}_j}{\theta_i}\right| \quad \text{where } i=1,2,....j \tag{4.28}$$

In Eq. (4.28), λ_K is the kth exact eigenvalue and θ_i is the ith computed eigenvalue. $Z_j^{(I)}$ is the jth element of vector $Z^{(I)}$. If ERROR(I) < RTOL, for I = 1,2 ..p (where RTOL is a user's specified tolerance, and p is the number of eigenpairs to be extracted) then the Lanczos iteration is considered to be converged and the program begins to perform the eigenvector transformation accordingly (see step 13 of Table 4.3).

4.3.2 The Lanczos Iteration Step by Step procedure

The Lanczos method can be summarized in a step by step algorithm as shown in Table 4.3 to obtain the lowest p eigen-pair solutions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Step 1.	Factorization : $K_\sigma = L\, D\, L^T$
	Form starting vector: $\gamma_o \neq 0$; $q_o = 0$
Step 2.	Compute: $M\,\gamma_o$
Step 3.	Compute : $\beta_1 = \sqrt{\gamma_o^T M \gamma_o}$; $q_1 = \frac{\gamma_o}{\beta_1}$
Step 4.	Compute : $P_1 = Mq_1$
	Lanczos iteration
	For j = 1,2,3, . . . , do
Step 5.	$\epsilon_j = K_\sigma^{-1} P_j$
Step 6.	$\delta_j = \epsilon_j - \beta_j q_{j-1}$
Step 7.	$\alpha_j = q_j^T M \delta_j = P_j^T \delta_j$
Step 8.	$\gamma_j = \delta_j - \alpha_j q_j$
Step 9.	$\Lambda_j = M\,\gamma_j$
Step10.	$\beta_{j+1} = (\gamma_j^T M \gamma_j)^{\frac{1}{2}} = \sqrt{\Lambda_j^T \gamma_j}$
	Reorthogonalization of q_{j+1}
Step11.	$q_{j+1} = \frac{\gamma_j}{\beta_{j+1}}$; $P_{j+1} = \frac{\Lambda_j}{\beta_{j+1}}$
Step 12.	IF necessary solve: $T_j z = \theta z$
	Converged? (If "No", then return to step 5)
Step 13.	Eigenvector transformation: $\phi = Q_j z$

Table 4.3: Step-by-Step Basic Lanczos Algorithm

4.3.3 Lanczos Iteration for positive definite systems: LDL^T

The step by step procedure in Table 4.3 for the basic Lanczos Algorithm has been coded for the solution of positive definite systems. All the matrix manipulations involved are performed using sparse technology. The system of equations in Step 1 of Table 4.3 is solved using the developed sparse solver for positive definite systems. Forward reduction and backward substitution are performed in Step 5 of Table 4.3. The efficient sparse matrix-vector multiplication is used throughout the algorithm.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A "predicted " eigen-value accuracy has been built inside the iterative Lanczos algorithm, and the "true" eigen-solution error norm is also calculated upon existing from the Lanczos iterative procedure, as shown in Eq.(4.19).

4.3.4 Lanczos Iteration for Indefinite systems: ODU-HKUST, ODU-MA27

In this section we extend the Lanczos algorithm to formulations that result in indefinite systems. The Lanczos eigensolver for indefinite systems that has been developed has the option of using either of the two sparse indefinite solvers presented in Chapter III, the ODU-HKUST and the ODU-MA27. For indefinite systems, the cause of failure happens in the solution of the system in (Eq. 4.21) or the first step of the Lanczos procedure. Of course for a system which is not indefinite, the tridiagonal system can be solved in double precision to reduce round-off errors. However, for very poorly-conditioned cases, the entire Lanczos process will fail if the solver is not robust.

To improve convergence of the eigenvalues, a spectral transformation of the original eigen problem is used. The implementation is simple if we substitute for the eigenvalue $\gamma_i = \rho + \mu_i$, with ρ a real number referred to as the shift.

4.4 Major computational tasks and Enhancements in Subspace iteration and Lanczos algorithm

Careful observations on the Subspace iteration, and Lanczos algorithms indicate that the following major computational tasks are required:

Major task 1: Matrix factorization (see step 2 of Subspace iteration, and step 1 of Lanczos algorithm).

Major task 2: Forward and backward equation solutions (see step 4 of Subspace iteration, and step 5 of Lanczos algorithm).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Major task 3: Matrix-Vector (or Matrix-Matrix) multiplications (see Steps 5, 6 & 8 of Subspace iteration, and Steps 2,4,7,9,10 & 13 of Lanczos algorithm).

Matrix factorization, forward & backward equation solution, and matrix-vector (or matrix-matrix) multiplications represent the major computational tasks for Subspace iteration, and Lanczos algorithms. Recent developments in sparse technologies [49] are fully utilized to improve the computational efficiency of both Subspace iteration, and Lanczos algorithms. In calculating the "true" eigen-solution error norm, efficient vectorized sparse matrix-vector multiplication scheme is used.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER V

INTERIOR POINT METHOD WITH POSITIVE AND INDEFINITE SPARSE SOLVERS FOR LINEAR PROGRAMMING PROBLEMS

5.1 Introduction

Optimization is concerned with achieving the best outcome of a given objective while satisfying certain restrictions. Mathematical programming problems may be classified into several different categories depending on the nature and form of the design variables, constraint functions, and the objective function. The linear programming describes a particular class of extremization problems in which the objective function and the constraint relations are linear functions of the design variables. Interest in linear programming has been intensified since Karmakar's publication in 1984 of an algorithm that is claimed to be much faster than the simplex method for practical and large-scale problems.

The standard mathematical formulation for linear programming problems consists of an objective function and a constraint set.

$$\begin{aligned} & Min\ \vec{c}^{\,T}\vec{x} \\ & subject\ to\ [A]\vec{x}=\vec{b} \\ & \vec{x}\geq 0 \end{aligned} \qquad (5.1)$$

where $\vec{c}$ and $\vec{x}$ are n x 1 vectors, [A] is an m x n matrix and b is an m x 1 vector. $\vec{c}^{\,T}x$ is referred as the objective function. The constraint set $[A]\vec{x} = b$ describes a feasible region in which the optimal solution $\vec{x}^{*}$ must lie. The general iterative solution process for optimization problems can be summarized as in Table 5.1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Step 1: Initial guess of the design variable: say $\vec{x} = \vec{x}^{o}$
Step 2: Find direction to travel: say $\hat{C}_p$
Step 3: Find step size, σ, along the direction $\hat{C}_p$
Step 4: New design : $\vec{x}^{(i)} = \vec{x}^{(i-1)} + \sigma \hat{C}_p^{(i-1)}$
Step 5: Check for convergence $\lVert \vec{x}^{(i)} - \vec{x}^{(i-1)} \rVert \le \epsilon$
- yes : stop
- no : Return to step 2.

Table 5.1 Step by Step solution process for optimization

5.2 Review of the simplex method

The main idea of the simplex method is to move from a vertex to a neighboring one where the cost is lower. After a finite number of steps, since there is only a finite number of corners of the feasible set, the cost is reduced as far as possible and the current vertex is optimal. A simplex step is really an exchange step, in which a zero component of $\vec{x}$ enters the basic group and a positive component leaves (it becomes zero) the basic group. There remains an important decision: which edge to choose? Starting with a given vertex that satisfies $Ax=b$ with only m nonzero components, there are n-m zero components that might be allowed to increase, and therefore n-m edges to select from. We choose an edge along which the cost drops as rapidly as possible.

It was noticed early in the history of linear programing that the cost coefficients could form a new row at the bottom of the matrix A and elimination could be applied to this row too. The bigger matrix is called a ***tableau***, and it contains all information about the linear programming problems as shown in Table 5.2. While the "simplex tableau" approach is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$\begin{bmatrix} A & B \\ \hline C & 0 \end{bmatrix}$$

Table 5.2 Simplex Tableau

useful for educational purposes, most (if not all) serious software has been coded based upon the "revised simplex" formulation.

The constraints in Eq.(5.1) can be also expressed in matrix notation as follows:

$$A\vec{x} = \vec{b} \quad or \quad [\,B\,,\,N\,] \begin{Bmatrix} \vec{x}_B \\ \vec{x}_N = \vec{0} \end{Bmatrix} = \vec{b} \quad \Rightarrow \quad \vec{x}_B = B^{-1}\vec{b} \qquad \textbf{(5.2)}$$

where B is a square matrix containing the columns of A that correspond to nonzero components of $\vec{x}$ (or $\vec{x}_B$), and N is a rectangular matrix that contains the remaining columns of A that correspond to $\vec{x}_N$. Similarly the objective function can also be partitioned as follows:

$$\vec{c}\vec{x} = [\vec{C}_B, \vec{C}_N] \begin{Bmatrix} \vec{x}_B \\ \vec{x}_N = \vec{0} \end{Bmatrix} \qquad \textbf{(5.3)}$$

and using Eq.(5.2)

$$\vec{c}\vec{x} = \vec{C}_B B^{-1}\vec{b} \qquad \textbf{(5.4)}$$

Premultiplying by B^{-1} on both sides of Eq.(5.2), we have

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$[I, B^{-1}N] \begin{Bmatrix} \vec{X}_B \\ \vec{X}_N = \vec{0} \end{Bmatrix} = B^{-1}\vec{b} \tag{5.5}$$

Since matrix B in Eq.(5.2) becomes identity matrix I, we do achieve the "canonical form" of the simplex tableau. If the zero components of $\vec{x}$ increase to some value x_N, then the nonzero components $\vec{x}_B$ must be reduced by $B^{-1}Nx_N$ in order to maintain equality in Eq.(5.5). Hence, the cost will be changed to

$$cx = c_B(x_B - B^{-1}Nx_N) + c_N x_N \tag{5.6}$$

Eq.(5.6) can be re-arranged to

$$cx = (C_N - C_B B^{-1}N)x_N + C_B x_B. \tag{5.7}$$

Thus

$$\vec{r} = C_N - C_B B^{-1}N \tag{5.8}$$

For minimization problems, if $\vec{r} \geq \vec{0}$ then current vertex is optimal, since $r_i x_{N_i} \geq 0$; thus, best decision is to keep $x_{N_i} = 0$ and stop. If some components of $\vec{r}$ are negative, then select the variable x (associated with the most negative component of $\vec{r}$) to enter the basic variable group.

After $\vec{r}$ is computed and entering (into basic) variable x_i is chosen , which component x_j should leave basic group? It will be the first to reach zero as x_i increases. (ratio b/a of simplex tableau). From Eq (5.5)

$$\vec{X}_B + B^{-1}N\vec{X}_N = B^{-1}\vec{b} \tag{5.9}$$

By taking a closer look at the product $B^{-1}Nx_N$ of Eq.(5.9)

$$[B^{-1}N]\vec{x}_N = \vec{v}^{\,i} x_i \tag{5.10}$$

where $\vec{v}^{\,i}$ is the ith column of $B^{-1}N$. Therefore, Eq.(5.9) becomes

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$\vec{x}_B + \vec{v}^i x_i = B^{-1}\vec{b} \tag{5.11}$$

the kth component of $\vec{x}_B$ will drop to zero when the kth components of $\vec{v}*x_i$ and $B^{-1}b$ are equal. This happens when x_i grows to:

$$x_i = \frac{k^{th}component \;\; of \;\; B^{-1}\vec{b}}{k^{th}component \;\; of \;\; \vec{v}^i} \tag{5.12}$$

Table 5.3 gives a classical step of the a simplex procedure.

Step 1	Compute $\vec{r} = C_N - C_B B^{-1} N$
Step 2:	If $r \geq 0$ stop; the current x is optimal. Otherwise find the most negative component r_i, and let the corresponding x_i increase from zero. (it is the *entering variable*). Let v be the corresponding column of $B^{-1}N$.
Step 3:	Compute the ratios in Eq.(5.12) , admitting only positive components of v. If the jth ratio is the smallest, then x_j is the *leaving variable.*
Step 4:	The new corner satisfies Ax=b with x_i now positive and x_j now zero. Compute this corner and by row operations in the tableau (or in the revised simplex) prepare for the next simplex step.

Table 5.3 A step of the simplex Method

5.3 Interior point methods

5.3.1 Introduction

Since the introduction of Karmarkar's method, there have been many variants of the method introduced. All these methods are based on the same basic concept and are referred to as interior point methods, IPM. The simplex method finds the solution to linear programming problems by moving along the boundary of the feasible region from one vertex to the next. This can create a large number of iterations. However, if we go through the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

interior of the feasible region, we can get to the optimal solution more efficiently. The idea is then to choose a starting point and move in the direction that improves the objective function as much as possible. Therefore the questions of concern are, at what point do we start and how far do we go? The choice of the starting or initial point is crucial. It is possible to implement Karmarkar's original idea of moving a near boundary point back to the center of a new simplex in several ways.

The key difference between the simplex method and the IPM is that the former will travel along the boundary of the feasible region (in order to find the optimal solution), while the latter will travel through the interiors of the feasible region. As we can see in Fig. 5.1, if we start at $\vec{x}_c$ (the center of the feasible region) and move in the direction of the gradient of the objective function, we can take a large step towards the optimum. However, if we

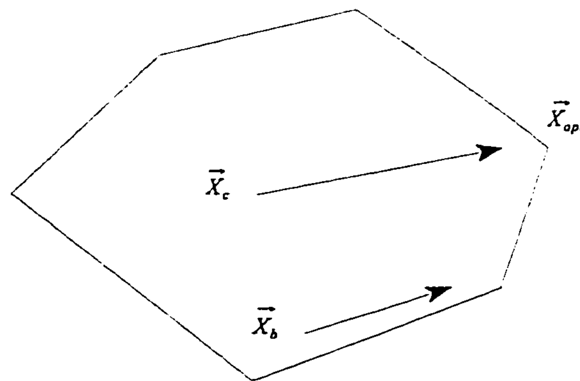


Fig. 5.1 Effect of the initial point on the step length

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

start at a point closer to the boundary, such as point $\vec{x}_b$, we can only take a very short step towards the optimum. The major drawback when applying gradient direction methods directly to the LP problem is that the objective function $C^T x$ always gives the same gradient direction no matter what the chosen point is. Therefore there will be only one step through the interior, and it will generally lead to nonoptimum point on the boundary (of the feasible region). Once on the boundary we are equivalently back to the simplex method.

To avoid the problem with the step size, Karmarkar had an ingenious idea, to take a step "almost" to the boundary. Thus, the point at which he stopped was still interior to the feasible region. Furthermore, from this new point he performs a variable (or a projective) transformation which will bring a point near the boundary (such as point $\vec{x}_b$) of the original simplex to near the center of the new simplex.

5.3.2 Variable transformation: Affine scaling method [62]

Assume, for the time being, that a starting point $\vec{x} = \vec{x}^o$, which is inside the feasible region, has already been found. A procedure, that will make sure that a feasible starting point $\vec{x}^o$ can be found, will be explained later. In order to overcome the difficulties of having the initial point close to the boundary, an affine scaling method is used. If

$$\vec{x}^o = [x_1^o, x_2^o, \ldots, x_n^o] \tag{5.13}$$

is the initial starting point, we define a diagonal scaling matrix D, and the following variable transformation is made:

$$\hat{x} = [D]^{-1}\vec{x} \tag{5.14}$$

where

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$[D] = \begin{bmatrix} x_1^o & & & \\ & x_2^o & & \\ & & \dots & \\ & & & x_n^o \end{bmatrix} \qquad \textbf{(5.15)}$$

From equation 5.14,

$$\vec{x} = [D]\hat{x} \qquad \textbf{(5.16)}$$

Thus, from the transformation in Eq. (5.14), the transformed coordinates of the starting iteration vector $\vec{x}_o$ are

$$\hat{x}^o = [1, 1, \ldots, 1] \qquad \textbf{(5.17)}$$

Substituting Eq. (5.16) into Eq. (5.1), the transformed problem can be reformulated as

$$\begin{aligned} &Min\ \hat{c}^T\hat{x} \\ &subject\ to\ \hat{A}\hat{x} = \vec{b} \\ &\qquad\qquad\quad \hat{x} \geq 0 \end{aligned} \qquad \textbf{(5.18)}$$

where

$$\hat{c}^T = \vec{c}^T D \qquad \text{or} \qquad \hat{c} = D\vec{c} \qquad \textbf{(5.19)}$$

and

$$\hat{A} = A D \qquad \textbf{(5.20)}$$

5.3.3 Direction of move $\hat{C}_p$

Since the new point $\hat{x}$ is already at (or close to) the center of the new simplex problem, one would like to take the steepest ascent (for maximization problem) direction and, at the same time, to remain inside the new (or transformed) feasible region (determined by Eq. 5.18). This projective direction will be referred to as $\hat{C}_p$ (see Fig. 5.2). To simplify

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the discussion assume that n=2 (or there are only 2 design variables), thus, the new (or transformed) feasible region can be shown in Fig. 5.2.

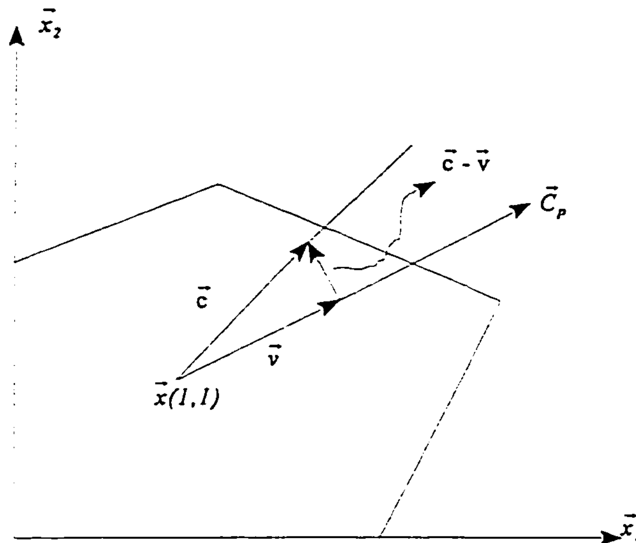


Fig. 5.2: Projective Steepest Ascent Direction

Let $\hat{x}_{new} = \hat{x} + \Delta\hat{x}$ be the new design variable. The new design variable still has to satisfy the constraints (such as Eq. 5.18)

$$[\hat{A}]\ (\hat{x} + \Delta\hat{x}) = \vec{b} \tag{5.21}$$

Using Eq. (5.18), then Eq. (5.21) becomes

$$[\hat{A}]\,\Delta\hat{x} = \vec{0} \tag{5.22}$$

Thus, $\Delta\hat{x}$ must be in the null space of $[\hat{A}]$. To find the projective direction $\hat{C}_p$, one needs to solve the following least square problem:

$$\begin{aligned} &Minimize && \frac{1}{2}(\hat{c} - \hat{v})^T(\hat{c} - \hat{v}) \\ &Subject\ to && [\hat{A}]\,\hat{v} = \vec{0} \end{aligned} \tag{5.23}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Eq.(5.23) has a physical interpretation, since the vector $(\hat{c}-\hat{v})$ represents the difference (or "error") between $\hat{c}$ and $\hat{v}$ (= same direction with $\hat{C}_p$), and naturally one would like to minimize the square of the "error". The constraint Eq.(5.23) is due to Eq. (5.22), since $\hat{v}$ plays the same role as $\Delta\hat{x}$.

The Lagrangian of Eqs. (5.23) can be computed as

$$L = \frac{1}{2}(\hat{c}-\hat{v})^T(\hat{c}-\hat{v}) + \lambda^T\hat{A}\hat{v} \tag{5.24}$$

Hence

$$\frac{\partial L}{\partial \vec{v}} = \vec{0} = \frac{1}{2}(-2\hat{c}+2\hat{v}) + \hat{A}^T\lambda \tag{5.25}$$

or

$$(\hat{v}-\hat{c}) + \hat{A}^T\lambda = \vec{0} \tag{5.26}$$

or

$$\hat{A}^T\lambda = (\hat{c}-\hat{v}) \tag{5.27}$$

Pre-multiply both sides of Eq (5.27), by $\hat{A}$, and utilizing Eq.(5.23), one obtains

$$\hat{A}\hat{A}^T\vec{\lambda} = \hat{A}\hat{c} \tag{5.28}$$

Equation (5.28) can be expressed as

$$[\hat{A}^*]\vec{\lambda} = \hat{C}^* \tag{5.29}$$

The dimension for $[\hat{A}^*], \vec{\lambda}$ and $\hat{C}^*$ in Eq. (5.29) are m x m, m x 1, and m x 1, respectively. Thus having found $\vec{\lambda}$ from Eq. (5.29) , one can compute the projective direction $\hat{C}_p$ (or $\hat{v}$) from Eq. (5.27) with the optimum solution of the least square problem, $\hat{v}^*$, equal to $\hat{C}_p$.

$$\hat{C}_p = \hat{c} - \hat{A}^T\vec{\lambda} \tag{5.30}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.3.4 Step size σ

Having found the appropriate search direction $\hat{C}_p$ ($\hat{C}_p$ = projected steepest ascend direction onto the null-space of $\hat{A}$) the question now is how far, σ, should we travel along the direction $\hat{C}_p$. The new design in the " scaled " design variable space $\hat{x}$ is :

$$\hat{x}_{new} = \hat{x}_{current} + \sigma\hat{c}_p \geq \vec{0} \tag{5.31}$$

or

$$\vec{1} + \sigma\hat{c}_p \geq \vec{0} \tag{5.32}$$

or

$$1 + \sigma\hat{c}_{pi} \geq 0 \quad \text{for } i = 1,2,\ldots,n \tag{5.33}$$

Each of Eqs.(5.32-5.33) must be satisfied to guarantee that $\hat{x}_i \geq 0$. For those positive components of $\hat{c}_{pi}$, Eq.(5.33) is automatically satisfied (since σ is a positive step size). However, for those negative components of $\hat{c}_{pi}$, Eq.(5.33) can be re-written as

$$1 - \sigma|\hat{c}_{pi}| \geq 0 \tag{5.34}$$

Hence

$$\sigma \leq \frac{1}{|\hat{c}_{pi}|} \tag{5.35}$$

Thus, to make sure that " All" components of $\hat{x}_{new} \geq 0$, we require:

$$\sigma_{\max} = \mathit{Minimun\ of}\ \{\hat{c}_{pi} \leq 0 : \frac{1}{|\hat{c}_{pi}|}\} \tag{5.36}$$

It should be noted there that if "all" $\hat{c}_{pi} \geq 0$, then we may select $\sigma_{\max}$ as large as we wish (in order to maximize the objective function) and still satisfy $\hat{x}_{new} \geq 0$. This is the case where the solution is unbounded. In order to avoid hitting the boundary of the feasible

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

region, a control parameter $\alpha = [0,1]$, say $\alpha = 0.98$ is introduced , so that Eq. (5.31) can be expressed as

$$\hat{x}_{new} = \hat{x}_{current} + \alpha\sigma_{\max}\hat{c}_p \tag{5.37}$$

Pre-multiplying both sides of Eq.(5.37) by [D], one has

$$[D]\hat{x}_{new} = [D]\hat{x}_{current} + \alpha\sigma_{\max}[D]\hat{c}_p \tag{5.38}$$

or

$$x_{new} = x_{current} + \alpha\sigma_{\max}[D]\hat{c}_p \tag{5.39}$$

The last issue which needs to be addressed in the section is how can we be sure to pick up a feasible starting point $\vec{x} = \vec{x}^{o}$.

5.3.5 Feasible starting iteration $\vec{x}^{o}$

Having introduced the slack, surplus and/or artificial variables, the design vector $\vec{x}$ can be partitioned into basic and non-basic variables. Thus, the constrained Eq. (5.1) can be expressed as

$$\sum_j a_{ij}^{B} x_j^{B} + \sum_j a_{ij}^{NB} x_j^{NB} = b_i \qquad for\ i=1,2,...,m \tag{5.40}$$

The coefficient matrix a_{ij}^{B} associated with the basic variable x_j^{B} is an identity matrix. Hence, Eq .(5.40) can be re-written as

$$x_i^{B} + \left(\sum_j a_{ij}^{NB}\right) x_j^{NB} = b_i \tag{5.41}$$

Now, let all the basic variables have the same *positive scalar* value x^{B}, and let all the non-basic variables have the same positive scalar value x^{NB} .

Then Eq. (5.41) can be expressed as

$$x_i^{B} = b_i - S_i x^{NB} \geq 0 \tag{5.42}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In Eq. (5.42), we have assumed $b_i \geq 0$ and S_i is the summation of all numerical values for the ith row of $[a_{ij}^{NB}]$. Thus

$$b_i \geq S_i x^{NB} \tag{5.43}$$

In Eq.(5.43), if $S_i < 0$, then this equation is guaranteed to be satisfied (since both b_i and x^{NB} are ≥ 0). However, if $S_i > 0$, then one obtains

$$x^{NB} \leq \frac{b_i}{S_i} \tag{5.44}$$

Thus, to make sure Eq. (5.44) is satisfied for any value of i, we will select x^{NB} as

$$x^{NB} = Minimum\ of \left\{ S_i > 0 : \ \frac{b_i}{S_i} \right\} \tag{5.45}$$

If Eq.(5.45) is enforced, then at least 1 of Eqs (5.44) will be " strictly" equal (i.e. $x^{NB} = \frac{b_i}{S_i}$). Thus, to be safer, a factor of $\frac{1}{2}$ is introduced, so that Eq. (5.45) becomes

$$x^{NB} = \frac{1}{2}\ Minimum\ of \left\{ S_i > 0 : \ \frac{b_i}{S_i} \right\} \tag{5.46}$$

Finally x_i^B, can be chosen according to Eq. (5.41)

$$x_i^B = b_i - S_i x^{NB} \tag{5.47}$$

The procedure explained in Eq.(5.13) through Eq.(5.39) constitutes the major steps of the optimizer to find the optimum solution given a feasible starting point. We call this Phase II. The IPM does not allow artificial variables in Phase II. In defining the starting iteration vector, a Phase I needs to be performed. In Phase I, iterations will be performed until all artificial variables are equal to zero. Thus Eqs. (5.46) and (5.47) will give the starting point for Phase I. Phase I will consist of minimizing the following problem:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$Min \sum_{i=1}^{NG+NE} Artificial\text{-}Variables_i \quad \text{subject to } [A]\vec{x}=\vec{b}, \quad \vec{x}\geq 0 \tag{5.48}$$

The procedure explained in Eqs (5.13-39) for phase II is also used to find the optimum solution of Phase I, which will be used as starting point for phase II of IPM. One may wonder why the artificial variables are not set equal to the right hand side (A_i=b_i) and other variables are set to zero, as the starting point for Phase I of the IPM. The reason is that the IPM will not accept it, since the IPM avoids to be on the boundary of feasible region (some variables = 0); that is why a factor α=0.98 has been introduced earlier in Eq.(5.37).

5.4 Step by Step Algorithm for the IPM

Following is the step by step algorithm for the IPM:

Step 1. Variable transformation

$$\hat{x} = [D]^{-1}\vec{x} \tag{5.49}$$

$$\hat{A} = [A]D \tag{5.50}$$

Step 2. Direction of search

$$\hat{C} = DC \tag{5.51}$$

$$\hat{A}\hat{A}^T\vec{\lambda} = \hat{A}\hat{c} \tag{5.52}$$

$$\hat{C}_p = \hat{c} - \hat{A}^T\vec{\lambda} \tag{5.53}$$

Step 3. Step size

$$\sigma_{max} = Minimun\ of\ \{\hat{c}_{pi} \leq 0 : \frac{1}{|\hat{c}_{pi}|}\} \tag{5.54}$$

Step 4. New design variable

$$x_{new} = x_{current} + \alpha\sigma_{max}\hat{c}_p \tag{5.55}$$

Step 5. Check for convergence

Table 5.4 Step by step algorithm for the IPM optimizer

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

5.5 Computational Enhancements and the Sparse Implementation of IPM

The implementation of IPM is performed in two phases that use the same phase II formulation. The first phase consists of finding the starting point iteration point that is in the feasible region and the second phase consists of finding the optimum solution. The optimum point of the first phase constitutes the starting point of the second phase. The constraint set given in Eq. (5.1) is input as a sparse matrix in a row wise unordered, and in NASA format (sparse unsymmetrical matrix). The input control parameter MREAD, allows the user to read the data in ASCII or binary form. Table 5.5 summarizes the algorithm implemented in the sparse IPM.

Step 1: INPUT DATA in NASA format

Step 2: Construct Slack, Artificial and Surplus Variable
Define the Basic set and the non-basic variables

Step 3: Construct the starting vector of phase I

Step 4: Phase I => call optimizer Table 5.4

Step 5: Phase II => call optimizer Table 5.4

Table 5.5 IPM algorithm

All the algebraic manipulation involved, in the step by step procedure given in Table 5.4 and 5.5 uses the sparse technology. The system of equation in Step 2 that arises from the IPM formulation can be solved using either the developed sparse solver for positive definite matrix or the indefinite solver. Both options are implemented and the choice depends on the properties of $\hat{A}\hat{A}^T$. The matrix $\hat{A}\hat{A}^T$ involves the multiplication of two sparse matrices given in row-wise format; one is the transpose of the other. A symbolic multiplication is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

performed before the numerical multiplication is completed. A counter of non-zero was inserted in the code to check the sparsity of the product $\hat{A}\hat{A}^T$.

Since matrix A is an augmented matrix made of the constraints set and a set of slack, artificial and surplus variables, it can be general by nature and one cannot guarantee that it will be positive definite. The matrix $\hat{A}\hat{A}^T$ of Eq. (5.52) often result in an indefinite system during the iterative process. For the example in Eq. (5.56), during the iterations, the

$$\begin{aligned} &Min\ Z = 2x_1 - 3x_2 \\ &subject\ to\ x_1 + 2x_2 \le 4 \\ &\qquad x_1 + 3x_2 \ge 6 \\ &\qquad x \ge 0 \end{aligned} \tag{5.56}$$

eigenvalues and eigenvectors of $\hat{A}\hat{A}^T$ satisfies: $\phi_1^T[\hat{A}\hat{A}]^T\phi_1 = \lambda_1$ and $\phi_2^T[\hat{A}\hat{A}]^T\phi_2 = \lambda_2$ and since there exist vectors ϕ_1 and ϕ_2 such that $\phi_1^T[\hat{A}\hat{A}^T]\phi_1 \le 0$ and $\phi_2^T[\hat{A}\hat{A}^T]\phi_2 \ge 0$; by definition $[\hat{A}\hat{A}^T]$ is indefinite. Therefore, in finding the solution for the direction of search in Eq. (5.52) , an indefinite solver may be required. An input control parameter. *ISOLVER*, specifies the type of solver to use, either the vector sparse solver for positive definite systems or the sparse solver for indefinite systems.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER VI

VECTOR-SPARSE SOLVER FOR UNSYMMETRICAL MATRICES

6.1 Introduction

Let's consider the following system of unsymmetrical linear equations

$$Ax = b \tag{6.1}$$

where the coefficient matrix A is unsymmetrical and the vectors x and b represent the unknown vector (nodal displacement) and the right-hand-side (known nodal load) vector, respectively. In Chapter II, we have developed a solver for symmetric positive definite systems. In this chapter, a solver for unsymmetrical matrices where the upper and lower triangular portions of the matrix are symmetric in location but unsymmetrical in value will be developed. Pivoting strategies for unsymmetrical matrices are not considered.

In order to take advantage of the algorithms discussed in Chapter II for the solution of symmetric matrices and exploit the vector capability provided by supercomputers, it is necessary to arrange the data appropriately. A mixed row-wise and column-wise storage scheme is used. This storage scheme offers the advantage of applying the symbolic factorization and the supernode evaluation only on one portion of the matrix instead of the entire matrix. Compared to the symmetrical case, the reordering (fill-in minimization), the numerical factorization, the forward/backward substitution and the matrix-vector multiplication subroutines are different since the matrix is unsymmetrical in values.

6.2 Sparse storage of the unsymmetrical matrix [67]

The unsymmetric matrix A is stored in a mixed row-oriented and column oriented fashion. The upper portion of the matrix is stored in a sparse, row-wise NASA format as it

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

has been explained in Section 2.2. The lower portion of the matrix is stored in a sparse column-wise format. Since a column-wise representation of a matrix is a row-wise representation of its transpose, and the matrix is symmetrical in locations, the array IA(neq+1), JA(ncoef), will be the same for both the upper and lower portion. AN(ncoef) will contain the coefficients of the upper portion of the matrix and a new array, AN2(ncoef), is introduced to store the coefficient values of the lower portion of the matrix. The diagonal values will be stored in the real array AD(neq). This storage scheme allows the use of the loop unrolling technique described in Chapter II during the factorization for both the upper and lower triangular portions of the matrix. Fig. 6.1 shows how the coefficient matrix A is stored.

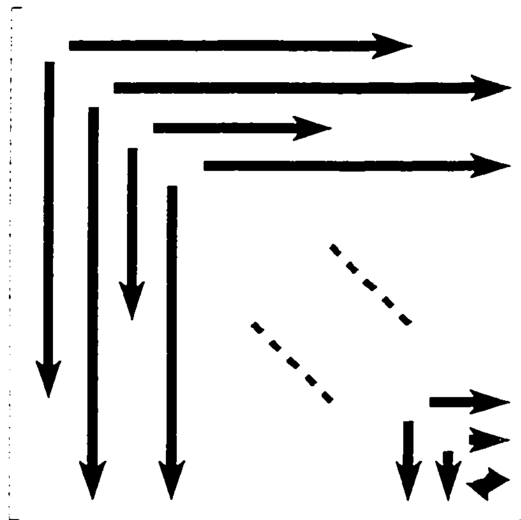


Fig. 6.1 Storage scheme for unsymmetrical matrix

To illustrate the usage of the adopted storage scheme, let's consider the matrix given in Eq.(6.2).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$A = \begin{bmatrix} 11. & 0. & 0. & 1. & 0. & 2. \\ 0 & 44. & 0. & 0. & 3. & 0. \\ 0 & 0 & 66. & 0. & 4. & 0. \\ 8 & 0 & 0 & 88. & 5. & 0. \\ 0 & 10 & 11 & 12 & 110. & 7. \\ 9 & 0 & 0 & 0 & 14 & 112. \end{bmatrix} \tag{6.2}$$

The data in Eq. (6.1) will be represented as follows

IA(1:7=*neq*+1) = {1, 3, 4, 5, 6, 7, 7 }

JA(1:6=*ncoef*) = {4, 6, 5, 5, 5, 6}

AD(1:6=*neq*) = {11., 44., 66., 88., 110., 112.}

AN(1:6=*ncoef*) = {1., 2., 3., 4., 5., 7. }

AN2(1:6=*ncoef*) = {8., 9., 10., 11., 12., 14. }

where *neq* is the size of the original stiffness matrix and *ncoef* is the number of non-zero, off diagonal terms of the upper triangular stiffness matrix (equal to the non-zero, off diagonal terms of the lower triangular stiffness matrix). Thus the total number of nonzeros off diagonal terms for the entire matrix is $2 \times ncoef$.

6.3 Basic unsymmetric equation solver

One way to solve Eq. (6.1) is first to decompose A into the product of triangular matrices, either LU or LDU. Since the graph of the upper and lower triangular matrices are the same, we chose the LDU factorization. Thus,

$$A = LDU \tag{6.3}$$

where U is an upper triangular matrix with unit diagonal, D a diagonal matrix and L a lower triangular matrix with unit diagonal. After factorization, the numerical values of matrix L are different from those of matrix U.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In order to better understand the general formula that we will derive for factorization of an unsymmetrical matrix, let's try to compute the factorized matrix [L], [D] and [U] from the following given 3x3 unsymmetrical matrix [A], assumed to be a full matrix in order to simplify the discussion.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \tag{6.4}$$

The unsymmetrical matrix A given in Eq. (6.4) can be factorized as indicated in Eq.(6.3) , or in the long form as follows

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} D_{11} & 0 & 0 \\ 0 & D_{22} & 0 \\ 0 & 0 & D_{33} \end{bmatrix} \begin{bmatrix} 1 & u_{12} & u_{13} \\ 0 & 1 & u_{22} \\ 0 & 0 & 1 \end{bmatrix} \tag{6.5}$$

The multiplication of matrices on the right-hand-side of the equality gives:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} d_{11} & d_{11}u_{12} & d_{11}u_{13} \\ l_{21}d_{11} & l_{21}d_{11}u_{12}+d_{22} & l_{21}d_{11}u_{13}+d_{22}u_{23} \\ l_{31}d_{11} & l_{31}d_{11}u_{12}+l_{32}d_{22} & l_{31}d_{11}u_{13}+l_{32}d_{22}u_{23}+d_{33} \end{bmatrix} \tag{6.6}$$

where the 9 unknowns (d_{11}, u_{12}, u_{13}, l_{21}, l_{31}, d_{22}, u_{23}, l_{32} and d_{33}) from Eq. (6.5) and Eq.(6.6) can be found by simultaneously solving the following system of equations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$
\begin{aligned}
a_{11} &= d_{11} \\
a_{12} &= d_{11}u_{12} \\
a_{21} &= l_{21}d_{11} \\
a_{13} &= d_{11}u_{13} \\
a_{31} &= l_{31}d_{11} \\
a_{22} &= l_{21}d_{11}u_{12} + d_{22} \\
a_{23} &= l_{21}d_{11}u_{13} + d_{22}u_{23} \\
a_{32} &= l_{31}d_{11}u_{12} + l_{32}d_{22} \\
a_{22} &= l_{31}d_{11}u_{13} + l_{32}d_{22}u_{23} + d_{33}
\end{aligned}
\quad (6.7)
$$

Thus from Eq. (6.7), one obtains

$$
\begin{aligned}
d_{11} &= a_{11} \\
u_{12} &= \frac{a_{12}}{d_{11}} \\
u_{13} &= \frac{a_{13}}{d_{11}} \\
l_{21} &= \frac{a_{21}}{d_{11}} \\
l_{31} &= \frac{a_{31}}{d_{11}} \\
d_{22} &= a_{22} - (l_{21}d_{11}u_{12}) \\
u_{23} &= \frac{a_{23} - (l_{21}d_{11}u_{13})}{d_{22}} \\
l_{32} &= \frac{a_{32} - (l_{31}d_{11}u_{12})}{d_{22}} \\
d_{33} &= a_{33} - (l_{31}d_{11}u_{13} + l_{32}d_{22}u_{23})
\end{aligned}
\quad (6.8)
$$

In solving for the unknowns in Eq. (6.8), the factorized matrices [L], [D] and [U] can be found in the following systematic pattern:

Step 1:The 1st diagonal value of [D] can be solved for d_{11}.

Step 2:The 1st row of the upper triangular matrix [U] can be solved for the solution of u_{12} and u_{13}.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Step 3:The 1st column of the lower triangular matrix [L] can be solved for l_{21} and l_{31}.

Step 4: The 2st diagonal value of [D] can be solved for d_{22}.

Step 5: The 2st row of the upper triangular matrix [U] can be solved for the solution of u_{23}.

Step 6: The 2st column of the lower triangular matrix [L] can be solved for l_{32}.

Step 7. The 3st diagonal value of [D] can be solved for d_{33}.

By observing the above procedure, one can see that to factorize the term u_{ij} of the upper triangular matrix [U], one needs to know only the factorized row i of [L] and column j of [U]. Similarly, to factorize the term l_{ji} of the lower triangular matrix [L], one needs to know only the factorized row j of [L] and column i of [U] as shown in Fig. 6.2.

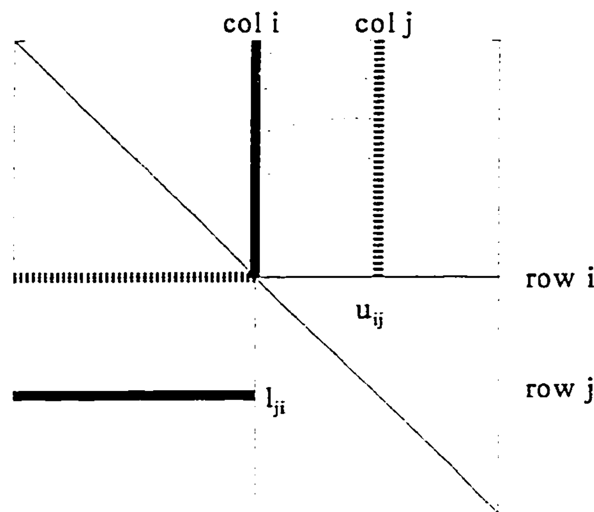


Fig. 6.2 Unsymmetrical solver: Factorization of u_{ij} and l_{ji}

By generalizing to a matrix of dimension *neq*, the ith row elements of [U] and the ith column elements of [L] can be obtained by the formulas in Eq.(6.9) and Eq.(6.10), assuming that the rows from 1 to i-1 and column from 1 to i-1 have already been factorized:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$u_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} l_{ik} d_{ii} u_{kj}}{d_{ii}} \qquad (j=i+1,neq) \tag{6.9}$$

$$l_{ji} = \frac{a_{ji} - \sum_{k=1}^{j-1} l_{jk} d_{jj} u_{ki}}{d_{jj}} \qquad (i=j+1,neq) \tag{6.10}$$

and the diagonal values will be given by Eq.(6.11)

$$d_{ii} = a_{ii} - \sum_{k=1}^{i-1} l_{ik} d_{ii} u_{ki} \tag{6.11}$$

Once the matrix is factorized, the unknown vector x is determined by the forward/backward substitution. Using Eq.(6.3) one can write Eq.(6.1) as follows:

$$LDy = b \tag{6.12}$$

with y= Ux. The solution of Eq. (6.12) can be obtained follows:

$$y_i^{*} = b_i - \sum_{k=1}^{i-1} L_{ik} y_k \quad (i=1,...,neq) \quad \text{with} \quad y^{*} = Dy \tag{6.13}$$

and to solve

$$Ux = y \tag{6.14}$$

for x,

$$x_i = y_i - \sum_{k=i+1}^{neq} U_{ik}\, x_k \qquad (i=neq,...,1) \tag{6.15}$$

The factorization is computationaly much more involved than the forward/backward substitution.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.4 Vector-sparse LDU unsymmetrical solver

6.4.1 Introduction

The vector-sparse unsymmetrical solver developed is a collection of subroutines that follow the same flowchart as the one given in Fig. 2.2, with the subroutines performing different tasks. Since the matrix is unsymmetrical in values, the reordering algorithm for symmetric matrix is not suitable. On the other hand, by observing Fig. 6.2 and the derivations in Eq. (6.3), the multipliers in the factorization of the upper portion of the matrix will be computed from the coefficients of the lower portion of the matrix and vice versa; thus, the numerical factorization will be different from the symmetrical case.

The purpose of symbolic factorization is to find the locations of all nonzero (including "fills-in" terms), off-diagonal terms of the factorized matrix [U]. Since both upper and lower portion of the matrix have the same graph, the symbolic factorization is performed only on either the upper or lower portion of the matrix. The symbolic factorization requires the structure IA, JA of the matrix in an unordered representation and generates the structure IU, JU of the factorized matrix in an unordered representation. However, the numerical factorization requires IU, JU to be ordered, while IA, JA can be given in an unordered representation. A symbolic transposition routine, TRANSA, which does not construct the array of non zero of the transpose structure, will be used twice to order IU, JU, after the symbolic factorization, since we are only interested in ordering JU. One of the major goals in this phase is to predict the required computer memory for subsequent numerical factorization for either the upper or lower portion of the matrix. For unsymmetrical case, the total memory required is twice the amount predicted by the symbolic factorization.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.4.2 Ordering for unsymmetrical solver

Ordering algorithms such as minimum-degree and nested dissection have been developed for reducing fill in factorizing sparse symmetric matrices. One cannot apply fill-in minimization, MMD (see Chapter II), on the upper and lower matrices separately. Shifting rows and columns of the upper portion of the matrix will require values from the lower portion of the matrix and vice versa. Let's consider the following example:

$$A = \begin{bmatrix} 100 & 1 & 2 & 3 & 4 \\ 5 & 100 & 6 & 7 & 8 \\ 9 & 10 & 100 & 11 & 12 \\ 13 & 14 & 15 & 100 & 16 \\ 17 & 18 & 19 & 20 & 100 \end{bmatrix} \tag{6.16}$$

Let's assume that the application of the Modified Minimum Degree (MMD) algorithm on the graph of the matrix results in the following permutation:

$$PERM\begin{Bmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{Bmatrix} = \begin{Bmatrix} 1 \\ 4 \\ 2 \\ 3 \\ 5 \end{Bmatrix} \tag{6.17}$$

By switching rows and columns of the matrix given in Eq. (6.16) according to the permutation vector PERM, given in Eq. (6.17), the reordered matrix A_r becomes

$$A_r = \begin{bmatrix} 100 & 3 & 1 & 2 & 4 \\ 13 & 100 & 14 & 15 & 16 \\ 5 & 7 & 100 & 6 & 8 \\ 9 & 11 & 10 & 100 & 12 \\ 17 & 20 & 18 & 19 & 100 \end{bmatrix} \tag{6.18}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

One the other hand, if one considers only the upper portion of the matrix (as for a symmetrical case), switching rows and columns of the matrix according to the permutation vector, PERM, will result in the following reordered matrix A_s given in Eq. (6.19). One can see that the elements A(2,3) and A(2,4) came from the lower portion. Therefore, rearranging the values of AN (or AN2) after

$$A_s = \begin{bmatrix} 100 & 3 & 1 & 2 & 4 \\ & 100 & 7 & 11 & 16 \\ & & 100 & 6 & 8 \\ & & & 100 & 12 \\ & & & & 100 \end{bmatrix} \quad \textbf{(6.19)}$$

the permutation vector PERM has been determined by the MMD routine will require certain elements of AN2 (or AN). The reordering subroutine for symmetric system has been modified to account for these changes and implemented without adding any additional working array. The portion of skeleton Fortran code in Table 6.1 shows how to retrieve efficiently the appropriate elements from the lower (upper) portion of the matrix, while constructing the reordered upper (lower) portion of the matrix. The permutation vector PERM and the structure IU and JU of the reordered matrix are assumed to be available already.

The algorithm in Table 6.1 is different for a case of a symmetrical matrix because, if only the upper portion of a symmetrical matrix is stored in memory, the numerical values in row i at the left side of the diagonal value are identical to the values in column i above the diagonal value (see Fig. 6.2). Consequently, the second DO loop 231 in Table 6.1 will not be needed because, all data can be retrieved from the upper portion of the matrix and one can

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
      DO 200 i=1, N-1
         I0=perm(i)
         D0 220 j=IU(i), IU(i+1)-1
         J0 = perm(JU(j))
         IF(I0.LT.J0) THEN
                  IJ0=I0
                  IJ00 =J0
                  DO 230 jj=IA(IJ0), IA(IJ0+1)-1
                     IF(JA(jj).NE.IJ00) GO TO 230
                     UN(j)=AN(jj)
                     UN2(j)=AN2(jj)
                     GO TO 220
             230  CONTINUE
         ELSE
                  IJ0=J0
                  IJ00 =I0
                  DO 231 jj=IA(IJ0), IA(IJ0+1)-1
                     IF(JA(jj).NE.IJ00) GO TO 231
                     UN(j)=AN2(jj)
                     UN2(j)=AN(jj)
                     GO TO 220
             231  CONTINUE
         ENDIF
     220 CONTINUE
200 CONTINUE
```

Table 6.1 Portion of Skeleton Fortran code of reordering of an unsymmetrical matrix

select the appropriate pointers IJ0 and IJ00 before the inner most DO loop. On the other hand, for an unsymmetrical matrix, one should scan separately the upper and lower portion of the matrix ($AN \neq AN2$) as shown in Table 6.1.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6.4.3 Sparse Numerical Factorization with loop unrolling

By observing Fig. 6.2 and the derivations in Section 6.3, in order to factorize an element u_{ij} of the upper triangular matrix, one needs to know the factorize row i of [L] and the column j of [U]. Thus, the multiplier of the upper portion of the matrix will be computed from the coefficient of the lower portion of the matrix. Table 6.2 give the pseudo Fortran skeleton code on how the multipliers are computed and how the factorization is carried out.

```
1.c........Assuming row 1 has been factorized earlier
2.         Do 11 I = 2, NEQ
3.         Do 22 K= Only those previous " master " rows which have contributions to
            current row I
4.c........Compute the multipliers
5.         XMULT = U(K,I) / U(K,K)
           XMULT2 = L(I,K) / U(K,K)
6.         Do  33 J = appropriated column numbers of " master" row # K
7.             U(I,J) = U(I,J) - XMULT2 * U(K,J)
               L(J,I) = L(J,I) - XMULT * L(J,K)
8.  33     CONTINUE
9.         U(K,I) = XMULT
           L(I,K) =XMULT2
10. 22     CONTINUE
11. 11     CONTINUE
```

Table 6.2: Pseudo FORTRAN Skeleton Code For Sparse LDU Factorization

In the sparse implementation, after the symbolic factorization is completed on one portion of the matrix, the numerical factorization requires IU, JU (structure of [L] or [U]) to be ordered and the required computer memory for the factorization is known.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Similar to the symmetrical case, the numerical factorization also requires to construct chain lists to keep track of the rows that will have contributions to the currently factorized row. Another advantage of the storage scheme that we have adopted is that the chain lists for the factorization of [L] (or [U]), will be the same as for the factorization of [U] (or [L]).

The loop unrolling strategies that have been successfully introduced earlier can also be effectively incorporated into the developed unsymmetrical sparse solver in conjunction with the master degree of freedom strategy. In the actual code implementation, "DO loops" in Eqs. (6.9 -6.11) will be rearranged to make use of loop unrolling technique. The loop unrolling is applied separately for the factorization of the upper portion and for the lower portion. Assuming the supernodes have already been computed (the supernodes of the upper portion is the same as the ones for the lower portion). The skeleton FORTRAN code in Table 6.2 should be modified as shown by the pseudo, skeleton FORTRAN code in Table 6.3 for a loop unrolling level 2.

6.4.4 Forward and Backward solution

The forward and backward solutions were implemented following the formula in Eqs.(6.12-6.15), once the factorized matrices [L], [D] and [U] are computed. In the forward solution , (Eqs. 6.12 and 6.13), the factorized matrices [L] and [D] are used, and in the backward substitution, the upper portion of the factorized matrix [U] is used.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
C .......Assuming row 1 has been factorized earlier
      Do 11 I=2, NEQ
      Do 22 K=Only those previous "master" rows which have contributions to
             current row I
C .......Compute the multiplier(s)
      NSLAVE DOF= MASTER (I) - I
      XMULT = U(K,I) / U(K,K)
      XMUL_m = U(K+m,I)/U(K+m,K+m)
      XMULT2 = L(I,K) / U(K,K)
      XMUL2_m = L(I,K+m)/U(K+m,K+m)
C .......m=1,2 ... SLAVE  DOF
      Do 33 J = appropriated column numbers of " master " row # K
      U(I,J) = U (I,J) - XMULT2 * U(K,J)  - XMUL2_m *U(K+m,J)
      L(J,I) = L(J,I) -  XMULT* L(J,K)  - XMUL_m *L(J,K+m)
  33  CONTINUE
      U(K,I) = XMULT
      U(K+m,I) = XMULm
      L(I,K) = XMULT2
      L(I,K+m) = XMUL2m
  22  CONTINUE
  11  CONTINUE
```

Table 6.3 : Pseudo FORTRAN Skeleton Code For Sparse LDU Factorization With Unrolling Strategies

6.4.5 Sparse unsymmetric matrix-vector multiplication

A matrix-vector multiplication subroutine has been efficiently designed for which the unsymmetrical matrix is stored in a mixed row-wise and column-wise storage scheme. The non zeros from the upper and lower triangular matrix are stored in two distinctive arrays AN and AN2 with the same structure IA and JA. Let's consider a vector *temp*(1:neq) that will contain the result of the matrix-vector multiplication. After multiplying the diagonal values by the right-hand-side, the multiplication of the upper and lower portion of the matrix are efficiently implemented as shown in Table 6.4.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
       DO 10 i=1,n
       iaa=ia(i)
       iab=ia(i+1)-1
       DO k=iaa, iab
       kk=ja(k)
       sum=sum+AN(k)*rhs(kk)
       temp(kk)=temp(kk)+ AN2(k)*rhs(i)
       ENDDO
       temp(i)=sum
    10 CONTINUE
```

Table 6.4 Unsymmetrical matrix-vector multiplication

The algorithm in Table 6.4 offers the advantage of avoiding to convert a row-wise complete unordered storage that is normally used for general unsymmetric matrix into our special storage scheme (mixed row and column-wise format).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER VII

APPLICATIONS

7.1 Introduction

The success of algorithms for sparse matrix computations depends crucially on careful computer implementations. All algorithms described in the previous chapters have been coded in standard Fortran 77, and therefore should port to other computer platforms with no or minor changes. The floating-point operations have been performed in double precision, except on the Cray Y-MP where single precision is used. On the machines with vector capability, all codes have been compiled with the vector optimization turned on to the optimum level (-O3 on most computers). The optimal level of loop unrolling varies from computer to computer. In our experiments, we have tried loop unrolling level-p (with p=1,2,4,and 8). All the test problems have been obtained from NASA Langley Research Center, except the Off-shore EXXON model [37-38,56]. All timing presented are in seconds.

The different computer platforms used in our experiments include (but not limit to) the following:

- Cray Y-MP from NASA Langley Research Center.

- IBM RS6000 model 590: A high performance computing workstation from the Office of Computing and Communication Service, OCCS, at Old Dominion University that we will refer to as *Stretch*.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- SUN workstations (Sparc 20 that will refer to as *Rhino*, a Sparc 10 and series of Sparc 5. We will refer to one SUN Sparc 5 as *Cedar*) from the civil engineering Unix laboratory at Old Dominion University.

- SUN workstation SPARC 20 that we will refer to as *USTSU31* from Hong Kong University of Science and Technology (HKUST).

- Silicon Graphics Indigo 2 from HKUST.

Unlike the SUN workstations, the Cray Y-MP has no cache memory. Its floating-point hardware is extremely fast due to vector pipelining. The use of loop unrolling, vector directives increase the gain in performance. It is also worth noting that Cray Y-MP machine performs floating-point arithmetic far more efficiently than integer arithmetic, in contrast to the workstations where the integer and floating-point performance is better balanced.

The IBM RS6000/590, *stretch*, from Old Dominion University is extensively used in the evaluation of the performance of the developed Fortran codes. It is a vector machine running the AIX XL Fortran compiler. The performance achieved on the *stretch* machine was not due to only the quality of the sparse algorithms, but also due to the selection of compiler options and flags. The following flag options were selected:

-bmaxdata:<bytes> : which specifies the maximum amount of space to reserve for the program data segment (if one needs more than 256 MB).

-bmaxstack:<bytes>: specifies the maximum amount of space to reserve for the program stack segment (if one needs more than 256 MB).

-O, -O2 : Optimizes code generated by the compiler.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

-O3: Performs the -O level optimizations and perform additional optimizations that are memory or compiler time intensive. The optimization level -O3 changes sometimes the semantic of the program.

-qstrict : Ensure that optimizations done by the -O3 option do not alter the semantics of the program.

-qalias=noaryovrlp : program does not contain array assignments of overlapping or storage associated arrays; can produce significant performance improvements for array language.

-qarch=pwr2 : produces an object that contains instructions that run on the POWER2 hardware platforms.

Each code is provided with a "makefile" that can port on different computer platforms. To compile most of the program, one just simply types "make". Porting from one computer to another typically requires minor changes to the makefile. To use a different computer platform, simply modify the makefile by commenting and uncommenting the appropriate script lines corresponding to the platform as it is described in the Appendix A. There are no calls to routines from external libraries. Only the timing subroutine, *cputime.f*, given in the Appendix B is machine dependent and must be modified when moving from one machine to another. The user may have to add timing calls for machines other than those currently studied. Currently covered are CRAY, SUN, IBM RS6000, and some other Unix boxes.

7.2 Description of various finite element models

In order to evaluate the performance (in terms of computational time, solution accuracy and memory requirements) of all the developed computer programs, we consider

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

applications that arise from practical finite element models. The following benchmark applications have been used to check the accuracy and robustness of all the developed computer programs.

7.2.1 Application No 1: High Speed Civil Transport (HSCT) Aircraft

The finite element model of the High Speed Civil Transport Aircraft, HSCT, shown in Fig. 7.1 and Table 7.1, resulted in a system of linear equations with 16,152 degrees of freedom and 373,980 nonzero off diagonal terms. Fig. 7.2 shows the sparsity pattern of the non zero elements of the upper part of the stiffness matrix.

7.2.2 Application No 2: The EXXON off shore model

The finite element model for the EXXON model (shown in Figs. 7.3-7.5 and Table 7.2) has been used extensively in earlier research works [37,38,56]. The resulted stiffness matrix has 23,155 degrees of freedom. The number of non-zero off diagonal terms of the original stiffness matrix is 809,427. Fig. 7.6 shows the sparsity patterns of the non zero elements of the upper part of the stiffness matrix.

7.2.3 Application No 3: Thermal-Structural model

The finite element model of the thermal-structural model resulted in a system of 43,806 linear equations with 1,037,705 non zeros coefficients of the stiffness matrix. Table 7.3 gives the characteristics of the finite element model, and Fig. 7.7 shows the sparsity patterns of the non zero elements of the upper part of the stiffness matrix.

7.2.4 Application No 4: Solid Rocket Booster (SRB)

The finite element model of the Solid Rocket Booster, SRB, shown in Fig. 7.8, resulted in a system of 54,870 linear equations with 1,308,185 nonzero off diagonal terms.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table 7.4 gives the characteristics of the finite element model and Fig. 7.9 shows the sparsity patterns of the non zero elements of the upper part of the stiffness matrix.

7.2.5 Indefinite matrices

In order to evaluate the performance (in terms of computational time, solution accuracy and memory requirements) of the proposed sparse solvers with pivoting strategies for symmetric indefinite systems, five NASA benchmarks problems (ranging from 51 to 15,357 unknown degree-of-freedoms) were considered in this study. The following applications are considered:

- Application No 5 : Cantilever Beam problem , 51 DOF.
- Application No 6 : Carlos Davilla problem, 247 DOF.
- Application No 7 : Jonathan's plate problem, 1,440 DOF.
- Application No 8 : Knight's panel problem, 2,430 DOF.
- Application No 9 : 15,367 DOF problem.

A summary of the characteristics of these five indefinite matrices are presented in Table 7.5. Fig. 7.10 to Fig. 7.14 give the sparsity patterns of the non zero elements of upper portion and the diagonal terms of the stiffness matrix.

- Application No 10: An additional application, the McDonell Douglas Stitched/RFI all composite wing finite element model with 53,948 degrees of freedom, is considered. The details of this model can be found in NASA TM 110267 by John Wang (or NASA TM 110267, by Wang, on NASA Langley Technical Report server). The finite element model contains 7,448 Quad elements, 2,562 Beam elements, 98 triangular elements and 24 NASA interface elements causing 4,326 zeros on the diagonal of the stiffness matrix. Fig. 7.15 shows the finite element model.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

7.2.6 Examples description for Interior Point Method (IPM)

To validate the accuracy and robustness of the developed interior point method, the following five small examples were considered that cover the different kinds of linear programming problems. Problems with a feasible region, no feasible solution, a feasible region as a point, unbounded and multiple solution are considered. Graphical solutions of these examples are also provided to check the accuracy of the IPM.

Application No 11 (optimum solution exist)

$$\begin{aligned} Min\ Z &= -2x_1 - 2x_2 \\ subject\ to\ 2x_1 + 3x_2 &\le 6 \\ 2x_1 + x_2 &\le 4 \end{aligned}$$

Application No 12 (Feasible solution is a point)

$$\begin{aligned} Min\ Z &= 2x_1 - 3x_2 \\ subject\ to\ x_1 + 2x_2 &\le 4 \\ x_1 + 3x_2 &\ge 6 \end{aligned}$$

Application No 13 (No feasible solution)

$$\begin{aligned} Min\ Z &= 3x_1 - 2x_2 \\ subject\ to\ 2x_1 + x_2 &\le 4 \\ 3x_1 + 3x_2 &\le 3 \end{aligned}$$

Application No 14 (Multiple solutions)

$$\begin{aligned} Min\ Z &= 2x_1 + 2x_2 \\ subject\ to\ 2x_1 + x_2 &\ge 4 \\ x_1 + x_2 &\ge 1 \end{aligned}$$

Application No 15

$$\begin{aligned} Min\ Z &= 2x_1 + x_2 \\ subject\ to\ 5x_1 + 10x_2 &\ge 8 \\ x_1 + x_2 &\le 1 \end{aligned}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Several other moderately large scale examples have also been formulated to check the performance of the developed IPM, as it will be described in the following paragraphs:

$$\begin{aligned} & Min\ \vec{c}^{\,T}\vec{x} \\ & subject\ to\ [A]\vec{x}=\vec{b} \\ & \qquad\qquad\quad \vec{x}\geq 0 \end{aligned}$$

where [A] is an unsymmetric matrix containing the constraints set. Matrix [A] is read in NASA row-wise format as a complete unsymmetrical matrix. The set of indefinite matrices provided in Section 7.2.5 are used as constraints (matrix [A]). An input parameter *mread* is added into the code. When *mread* is equal to -1 , only the upper triangular part of matrix [A] is read and when mread is 1, the lower portion is also considered. The objective function is defined as the summation of all the design variables, $\vec{c}^{\,T}$= [1,1, ..., 1]. The design variables are assumed to be positive.

	Number of constraints	mread
Application 16	51	-1
Application 17	51	1
Application 18	247	-1
Application 19	247	1
Application 20	1440	-1

7.3 Numerical Results

All numerical results for the above 20 applications will be reported in this section.

7.3.1 Sparse equation solvers

a) LDL^T numfa1/2/8

The High Speed Civil Transport aircraft, the Exxon model, the thermal-structural problem and the Solid Rocket Booster finite element models are used to check the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

performance and robustness of the developed vector sparse LDL^T solver. Except the thermal-structural problem, which is non-positive definite, the stiffness matrix of all the other finite element models are positive definite. To check the accuracy of the results, an absolute error norm and relative error norm have been computed as follows:

$$\text{Absolute Error Norm} \qquad AEN = \|Kx - f\| \tag{7.1}$$

$$\text{Relative Error Norm} \qquad REN = \frac{\|Kx - f\|}{\|f\|} \tag{7.2}$$

where [K], {x}, and {f}, shown in Equations (7.1) and (7.2), correspond to the coefficient matrix, unknown vector and the right-hand-side vector, respectively. Table 7.6 and Fig. 7.16 give the numbers of non zeros after factorization and memory requirements for the HSCT application with different reordering schemes. The Nested Dissection (ND) algorithm results in 13.2% fill-in reduction and 18.5% for the Multiple Minimum Degree (MMD) algorithm on the HSCT finite element model (application No 1). The MMD seems to minimize the fill-in quite efficiently and requires less memory. Table 7.7 -7.8 shows the performance of Numfa1/2/8 for different level of loop unrolling using MMD on the HSCT finite element model. Table 7.9-7.10 shows the summary of all results for different reordering schemes and different level of loop unrolling on *Rhino* and *Stretch* machines. Figures 7.17 and 7.18 compare the factorization and total time of NUMFA1, NUMFA2 and NUMFA8 for the HSCT finite element model respectively on *Stretch* and *Rhino* machine. The following notations are used:

- Reord : reordering
- Loop unrol : Loop unrolling
- Symfa : symbolic factorization

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Numfa : numerical factorization

- FBE : Forward/backward solution

The total time given in Tables 7.9 and 7.10 does not include the time for the reordering. It is the overall time to read data from the disk (after reordering of the matrix is done), plus the time to perform the symbolic factorization, the transposition of the structure, the numerical factorization and the error norm-check. One can notice that the MMD with loop unrolling level 8 gives the best timing for the numerical factorization. Table 7.11 to 7.13 gives the comparison of results for the EXXON, Thermal-Structural and SRB finite element models, respectively, using MMD and different level of loop unrolling on the IBM R6000/590 (*stretch)* machine.

The IBM RS6000/590 (*Stretch*) has flag options for the vector compiler to enhance the performance. Figures 7.19 and 7.20 shows the impact of the compiler optimization level on the numerical factorization and total time for the HSCT and SRB finite element models respectively. Compiler optimization level -O2 and -O3 can give up to 76.4% gain in performance for the numerical factorization and up to 75.4% gain in performance for the total time for the applications that we have tested. To achieve a good performance, one should not only fine tune his algorithm implementation but also have a good knowledge of a particular computer platform.

Since most of the computer platforms that we have been using are not in dedicated environment (multi-users environment), most of the results have been recorded late at night (after 2:00 am) to try to have nearly dedicated time. Further testing have been done on the *Rhino* and *stretch* machine to see how reliable the time function is. The HSCT finite model has been used for studying various time functions, and NUMFA8 solver for positive definite

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

systems has been executed twenty times on each machine. The numerical factorization and the total time have been recorded. The statistical software, SAS was used to analyze the data, and the results can be summarized as follows:

	Rhino HSCT-Numfa	*Rhino* HSCT-Total	*Stretch* HSCT-Numfa	*Stretch* HSCT-Total
Mean	252.9595	287.6132	16.9630	20.2235
Variance	1.3008	1.7947	0.0266	0.0336
Standard Deviation	1.1405	1.3397	0.1631	0.1834
Standard mean	0.2550	0.2996	0.0365	0.0410
Maximum	255.7281	290.6975	17.4500	20.8400
Minimum	251.8622	286.3397	16.8400	20.0700
Range	3.8659	4.3578	0.6100	0.7700
Skewness	1.2634	1.2585	2.4945	2.7082

The time function on the IBM RS6000/590, *Stretch*, is more reliable than the one on the SUN SPARC 20, *Rhino*.

Table 7.14 shows an example of input data file, K.INFO, for the developed solver, NUMFA1/2/8 and Table 7.15 gives an example of an output file from the sparse solver NUMFA8. The following control parameters are considered in the input data file K.INFO:

- nreord : Reordering algorithm

  = 0 : No reordering scheme

  = 1 : Reverse Cuthill-McKee (RCM)

  = 2 : Nested dissection (ND)

  = 3 : Modified Minimum Degree (MMD)

-loop : Loop unrolling level

  = 1 : numfa1 : level 1

  = 2 : numfa2 : level 2

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

= 8 : numfa8 : level 8

-neq : number of equations

-ncoef : number of non zeros

-mread : input data

= -1 : read K.* NASA input files

= else : Read fort.* files

b)Cholesky OakRidgeODU

The first four applications have also been used to check the performance of the OakRidgeODU solver. Since this solver uses Cholesky algorithm, the non-positive definite thermal-structure problem has been modified, by imposing a large diagonal value to make it become positive definite. Tables 7.16 and 7.18 show the impact of the cache size on the HSCT Finite element model on the *stretch* and *Rhino* machines. A cache size of 64 and 32 gives the best performance on *stretch* and *Rhino* machines, respectively. Table 7.17 and 7.19 show the impact of the loop unrolling level on the performance of the solver on the *stretch* and *Rhino* machines. For different level of loop enrolling, the best performance has been achieved at level 4 and 8. Similarly, Tables 7.20-7.25 summarize the impact of cache size and loop unrolling level on the EXXON, Thermal-Structural and SRB finite element models.

Table 7.26 gives an example of an input data file, K.INFO, to run the OakRidgeODU solver and Table 7.27 gives an example of an output file from this solver. The following control parameters are considered in the input data file K.INFO:

- icase : ordering choice

= 1 natural

= 2 multiple minimum degree

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- cachsz: machine cache size (in Kbytes), usually 0, 32 or 64
- level : level of loop unrolling (1,2,4,and 8)
- neq : number of equations
- ncoef : number of non zeros
- mread: input data

    = -1 : read K.* NASA files

    = else : Read fort.* files

c)ODU-HKUST indefinite solver

The benchmark indefinite matrices of applications No 5 to No 10 provided by NASA Langley Research Center, are considered to evaluate the performance of the developed indefinite solvers. All these applications have a similar characteristic, they all use NASA interfaced elements, which cause zero terms on the diagonal of the stiffness matrix (refer to Figs. 7.10-7.14). Table 7.28 also gives the number and percentage of diagonal zero values. The total number of equations (or the number of degree of freedom) and the total number of nonzero coefficients before (ncoef) and after (ncoef2) factorization are also shown in Table 7.29. The relative Error Norm (REN) is computed according to the formula given in Eq.(7.2).

Further improved performance was achieved on the ODU-HKUST, by applying the MMD re-ordering algorithm (to minimize the fills-in terms) and by moving all zero diagonal terms of the original stiffness matrix toward the bottom right of the original stiffness matrix. Table 7.31 shows the gain achieved by using MMD and pushing the rows/columns corresponding to zero diagonal terms to the end, compared to the case where MMD is applied alone. Approximately 58% gain in performance has been achieved on the numerical

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

factorization of application No 9. Both Cray-YMP (single processor) computer and the IBM-RS6000/590 workstation are used in this study. For structural examples considered in this section, the resulting linear system of indefinite equations, shown in Eq. (2.1) can be expressed in the following form

$$\begin{bmatrix} A & B \\ B^T & 0 \end{bmatrix} \begin{Bmatrix} X \\ \lambda \end{Bmatrix} = \begin{Bmatrix} b \\ c \end{Bmatrix} \qquad \textbf{(7.3)}$$

In equation (7.3), the vector {X} can be referred to as the "displacement" vector, where as the vector $\{\lambda\}$ (which corresponds to the zero diagonal terms of the coefficient stiffness matrix) can be referred to as the "Lagrange multiplier" vector. The bottom right submatrix of the coefficient stiffness matrix, shown in Eq.(7.3), is a "zero" submatrix. Table 7.28 gives the percentage of zero diagonal values for all the indefinite matrices. The relative " displacement & Lagrange multiplier" error norm (or R.E.N) has been calculated, according to Eq. (7.2).

Golub [6] has suggested to use the value for the control parameter alpha, $\alpha = (1 + \sqrt{17})/8$. In our code, this value has been used as an input parameter. Figure 7.21 shows the impact of the choice of the control parameter alpha on the performance of the solver on the application No 9. Table 7.32 also gives the impact on the number of two-by-two (2×2) and diagonal interchange (one-by-one pivoting), as well as the non-zeros after fill-in (due to the choice of the control parameter alpha). Up to 79.4% gain can be achieved in the numerical factorization of application No 9.

Comparisons given in Table 7.30 have been made based upon structural data and compared to the results from the Boeing indefinite solvers for applications No5 to No9. The comparison has been made based on several different criteria

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(a) The maximum displacement

(b) The absolute summation of the entire DISPLACEMENT vector. As an example: assuming the DISPLACEMENT vector is { 1.2, -2.6, 0.7, 2.9}, then the maximum displacement is 2.9, and the summation (absolute) of all displacements is 1.2 + 2.6 +0.7 + 2.9

(c) The Relative Error Norm (REN) considered in solving the system [A]*{x} = {b} is defined in Eq.(7.2)

The ODU-HKUST solver performs well on matrix of size less than 15,367 but it is slow on large size matrix such as application No 10.

d)ODU-Ma27 indefinite solver [66]

The benchmark indefinite matrices given in application No 5 to No 10 are used again to evaluate the performance of ODU-Ma27 indefinite solver. Table 7.33 and 7.34 give a summary of results on *Rhino* and *stretch* machines. The relative error norm has been computed according to Eq.(7.2). The maximum and summation of the absolute value of the displacement, plus the lagrange multiplier, as well as the one for the displacement alone are shown in Table 7.33 and 7.43.

7.3.2 Sparse eigen-solvers

a)Lanczos and Subspace sparse eigensolvers for positive definite matrix

Based upon the discussions in previous sections, practical finite element models (such as Exxon-off-shore structure, and High Speed Civil Transport Aircraft) are used to evaluate the performance of the developed sparse eigen-solvers for positive definite systems that we called SPARSEPACK. Since the codes have been written in standard FORTRAN language (and without using any external library subroutines), it can be ported

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to different computer platforms (such as SUN SPARC 20, IBM-R6000/590, Intel Paragon, Cray C90 etc...) with no (or minimum) changes to the codes. The accuracy of the developed codes for solving generalized eigen equations can be measured by the Relative Error-Norm (=R.E.N.) which can be computed as :

$$\text{R.E.N.} = \frac{\|K\phi - \lambda M\phi\|}{\|K\phi\|} \tag{7.4}$$

The basic Subspace iteration code, that we will refer to as KJBATHE96, given in Ref. [1], will be used as a based-line reference. This basic Subspace iteration code [1] will be compared to the developed basic, "sparse" Subspace iteration (option also referred to as SVSub), and "sparse" Lanczos (option also referred to as SVLan) codes. For a fair comparison, the KJBATHE96 code is also compiled using the vector compiler on the IBM Stretch machine. Lumped masses are used in all examples in this section, but the Fortran code developed also has the capability to solve consistent mass matrix. In order to accelerate the calculations of the required eigensystem and avoid the singularity associated to systems with rigid body modes, the option of using a shift factor (see Eq.(4.5)) is implemented. The SPARSEPACK package contains not only the Subspace iteration and the regular Lanczos iteration, but also the block Lanczos (block less than 4).

The finite element model for the HSCT aircraft (see Fig. 7.1 and Fig. 7.2) has been used extensively in earlier research works. The numerical performances of 3 generalized eigen-solvers (KJBATHE96, Subspace iteration and Lanczos iteration) are presented in Figs. 7.22-7.23.

The finite element model for the EXXON model (see Fig. 7.3-7.6) used extensively in earlier research works [37,38,56]. The resulted system of generalized eigen-equations

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

from the EXXON model has 23,155 dof. The numerical performances are summarized in Figs. 7.24-7.25. It should be noted here that on the IBM-RS6000/590 workstation, vector processing capability is available, where as the vector processing capability is "not" available on the Sun SPARC 20 workstation (USTSU31).

Table 7.35 shows an example of input data file, K.INFO, for the developed eigen-solver package SPARSEPACK. The following control parameters are considered in K.INFO:

-nord : Reordering algorithm

= 0 : No reordering scheme

= 3 : Modified Minimum Degree (MMD)

-neig : number of required eigenvalues

-lump : Lump or consistent mass

= 1 : lump mass

= else : consistent mass

-neq : number of equations

-ncoef : number of non zeros

-ishift : shift

= 0 : no shift is considered

= else : shift is considered

-iblock :

= -1 : Subspace Iteration

= 0 : Regular Lanczos

= 1, ... ,3 : Block Lanzos (block 1, .., 3)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(The value of iblock has to be less than 4)

-mread : input data

= -1 : read K.* NASA input files

= else : Read fort.* files

Table 7.36 gives an example of output files from the eigensolver using Lanczos. Table 7.37 gives an example of output files using Subspace, and Table 7.38 gives an output of the KJBATHE96.

b) Lanczos and Subspace sparse eigensolver for Indefinite systems

Lanczos and Subspace iteration for indefinite systems have been implemented that uses the two indefinite solvers discussed in Chapter III (the ODU-HKUST indefinite solver and the ODU-Ma27 indefinite solver). Therefore, two codes have been developed for Lanczos and Subspace iteration, using both the indefinite solvers. A flag *imethod* is considered that takes the value 1 when the ODU-Ma27 indefinite solver is used, and the value 2 when the ODU-HKUST indefinite solver is used. Additionally, both lump and consistent mass can be treated. Finally, to shift the spectrum of eigenvalues and accelerate the convergence of the required eigensystem and avoid the singularity associated to systems with rigid body modes, the option of using a shift factor according to Eq. (4.5) has also been implemented. These different options have been implemented in different modules for a better memory management.

The accuracy has been measured by computing the Relative Error-Norm (=R.E.N) defined in Eq.(7.4). The indefinite systems in applications 5 to 9 have negative and positive eigenvalues. Table 7.39 gives an example of 15 eigenvalues of application No 6. (247 dof indefinite matrix). The following observations can be made:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- The Subspace iteration was able to capture both negative and positive eigenvalues, but the Lanczos gave the lowest positive eigenvalues (if no shift factor is considered). Table 7.39 and 7.40 shows an example of 15 eigenvalues computed from Subspace and Lanczos algorithms for the 247 DOF application

- The use of a shift factor will help to accelerate the convergence, and to handle systems with rigid body modes (but shift the spectrum of eigenvalues around the shift value).

Table 7.41 shows an example of an input data file, K.INFO, and Table 7.42 gives an example of a typical output file. The following control parameters are considered in K.INFO:

-neig : number of required eigenvalues

-lump : Lump or consistent mass

= 1 : lump mass

= else : consistent mass

-neq : number of equations

-ncoef : number of non zeros

-ishift : shift

= 0 : no shift is considered

= else : shift is considered

-mread : input data

= -1 : read K.* NASA input files

= else : Read fort.* files

We have developed robust sparse package for the eigensolution of positive-negative and indefinite symmetric matrices. Two challenging problems have been given to us by

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

NASA Langley Research Center to validate our code. Descriptions of these 2 problems are given in the following paragraphs:

1.- The Jonathan's ill-conditioned problem: An ill-conditioned stiffness matrix collected from a finite element procedure with 900 degrees of freedom and 11989 non-zeros off diagonal coefficients has been obtained. Table 7.43 shows the results provided by NASA test bed for the first 25 eigenvalues, and Table 7.44 and 7.45 give the results from our Lanczos and Subspace eigensolver, respectively.

2.- NGST Satellite Model: 5156 dof problem: This problem has 5156 dof and 88966 non-zeros off diagonal coefficients. The stiffness matrix contains some rigid body modes. It took 51 sec (time also includes reading data and error norm check) on the stretch machine to solve for the first 100 eigenvalues. The output is given in Table 7.46. A shift value was needed to deal with the singularity of the stiffness matrix. The first six eigenvalues are zeros (rigid body modes) and some repeated eigenvalues have been observed in the output (26th and 27th eigenvalues, 56th and 57th eigenvalues, etc).

7.3.3 Interior Point Method

Based upon the IPM and the indefinite sparse solver algorithms described in Chapters III and V, a Fortran computer code has been written to validate the entire numerical procedure. All results in this section have been obtained using the *cedar* computer (Sun SPARC 5) at Old Dominion University, and presented in Table 7.47 and 7.48, where, NEQ, NCOEF and NCOEF2 are Number of Equations, number of non-zero off diagonal coefficients of matrix $[AA^T]$ and number of non-zero off diagonal coefficients of matrix $[AA^T]$ including the diagonal values, respectively.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The first five small-scale examples (see Table 7.47) are used to validate the IPM code for different type of problems, such as feasible region is defined, feasible region is a point, no feasible region, multiple solutions. Fig. 7.26 to 7.30 give their graphical solutions. The last five medium-scale examples (see Table 7.48) are used to evaluate the numerical (by measuring the time) performance of the IPM, in conjunction with the developed indefinite sparse solvers.

Table 7.49 shows an example of input data file, K.INFO, for the developed IPM and Table 7.50 gives an example of output files from the solver. The following control parameters are considered in the input data file K.INFO:

-nv : number of design variables

-nl : number of inequality constraints (less than zero)

-ng : number of inequality constraints (greater than zero)

-ncoef : number of non-zeros in the constraint set

-isolver: type of solver used

= 1 : sparse solver for positive definite systems

= else: sparse solver for indefinite systems

-mread : input data

7.3.4 Sparse unsymmetrical solver

Three examples are considered to evaluate the performance of the developed unsymmetrical vector sparse LDU solver (that we will refer to as UNSYNUMFA). Two applications, the HSCT (16,152 degree of freedoms) and the SRB (54,870 degrees of freedoms) finite element models for which the static solution is known are considered. Another application , PierrotHSCT (16,152 degree of freedoms) is constructed by

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

considering the structure of the HSCT FEM with the same coefficient values for the upper portion of the matrix and different values for the lower portion of the matrix to make the matrix completely unsymmetrical in values.

To check the accuracy of the results , a relative error norm is computed as shown in Eq. (7.2), where matrix [K] is unsymmetrical. The sparse unsymmetrical matrix-vector multiplication subroutine developed in Section 6.4.5 is used to compute the product [K].{x} (where {x} is the displacement vector), which is required for error norm computation.

Table 7.51 gives the number of non-zeros and memory requirement for the HSCT FEM application with and without calling the subroutine for ordering unsymmetric matrix (UnsyMMD), explained in Section 6.4.2. By comparing the results in Table 7.51 to the symmetrical case in Table 7.6 for the HSCT application, the number of fill-in doubles but the total memory needed increases by 49.2 %. The use of reordering, UnsyMMD, decrease the non-zeros off diagonal by 18.5 % after factorization (as shown in Fig. 7.31) and 16 % in saving for the total memory needed by the solver.

Table 7.52-7.53 and Fig. 7.32-7.33 give a summary of results for different level of loop unrolling on the IBM RS6000/590 *stretch* with and without using the reordering (UnsyMMD). Table 7.54 and Fig. 7.34 give the summary of results for PierrotHSCT application and Table 7.55 and Fig. 7.35 give the summary of results for the SRB example.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

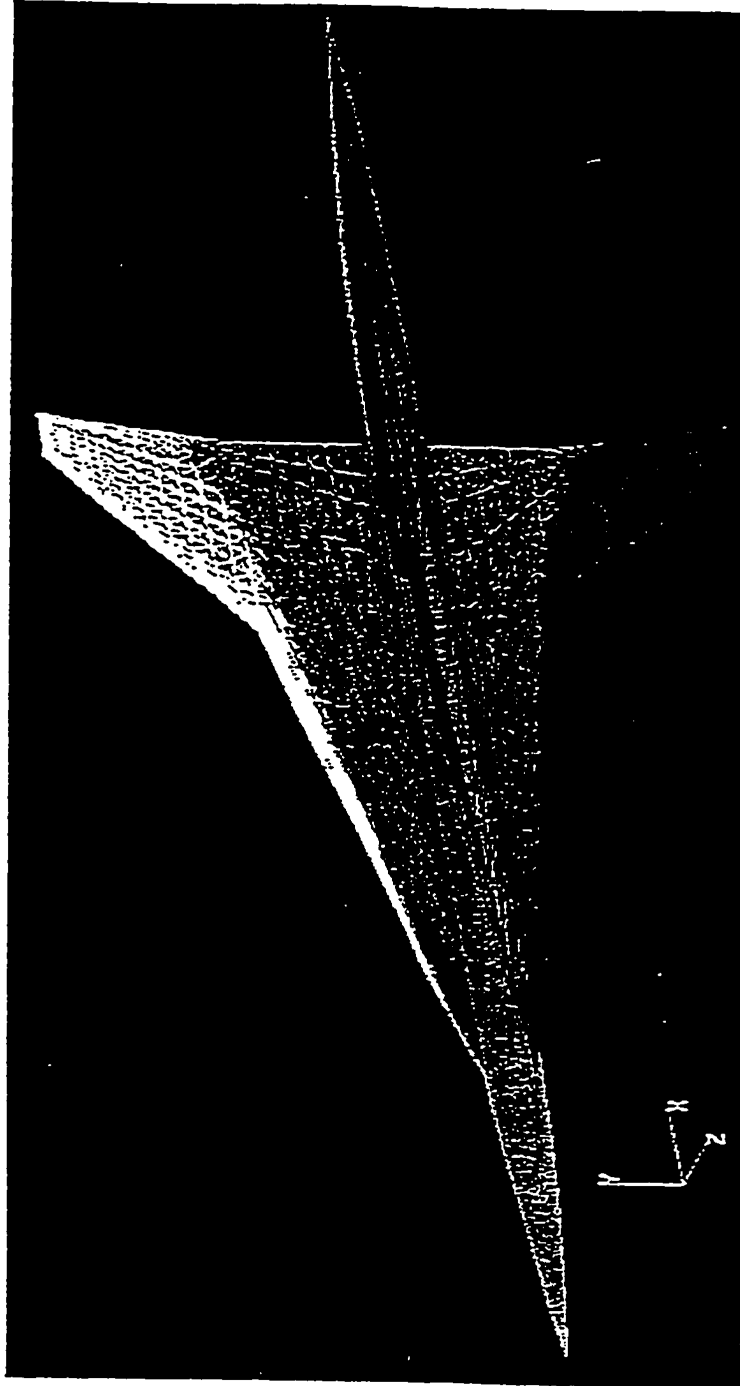


Fig. 7.1 High Speed Civil Transport aircraft, HSCT.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION No 1

Equations	Coefficients	Maximum Semi-bandwidth	Average Semi-bandwidth
16,146	499,505	593	318

Table 7.1 Characteristics of the NASA High Speed Civil Transport Aircraft FEM

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION N° 1

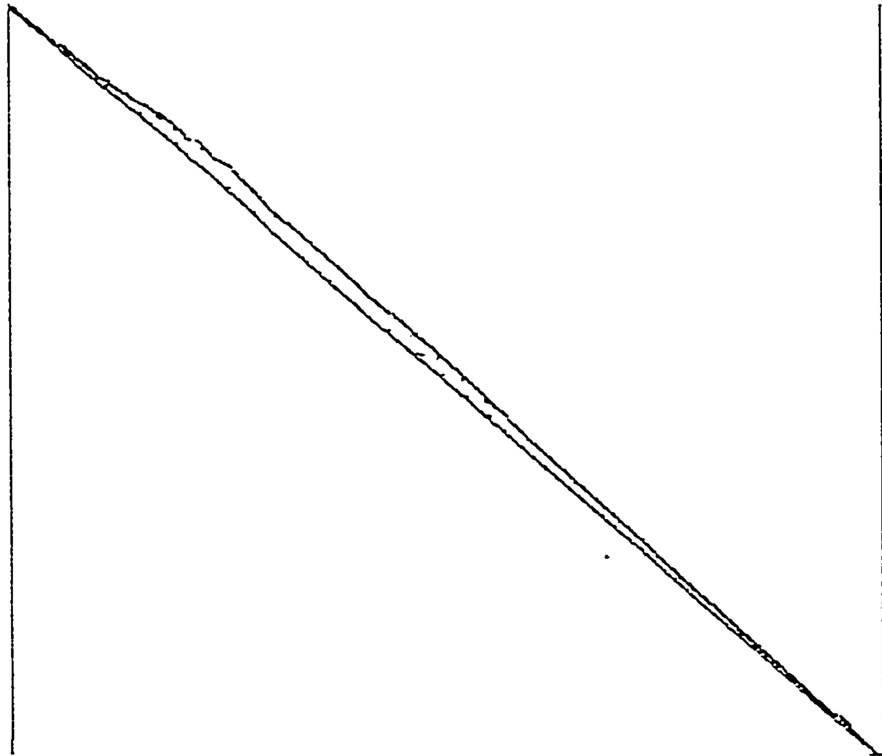


Fig. 7.2 Non-zero pattern of the NASA High Speed Civil Transport Aircraft FEM

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION N° 2

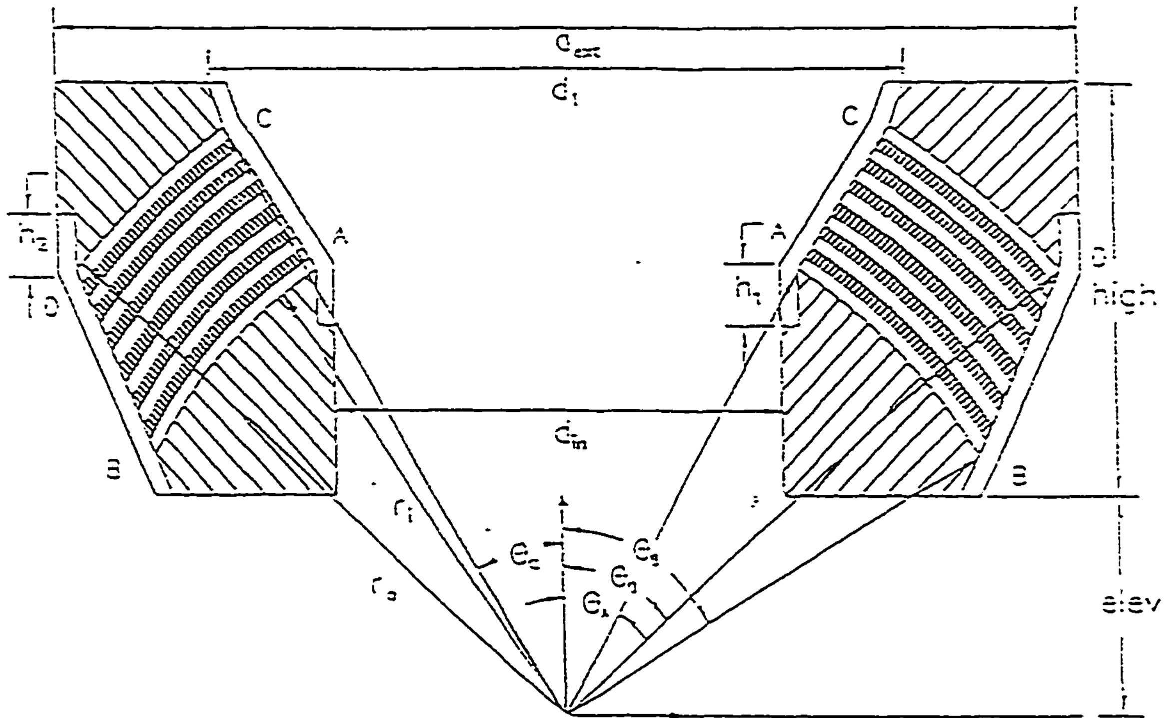


Fig. 7.3 TLP Flexjoint Geometry Parameters of the EXXON FEM

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION N° 2

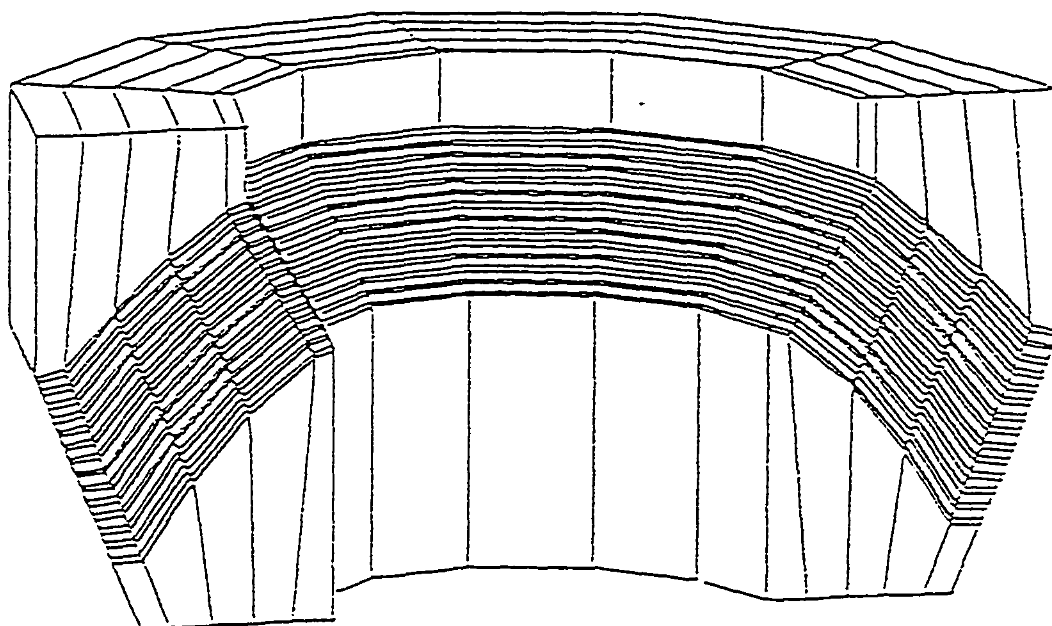
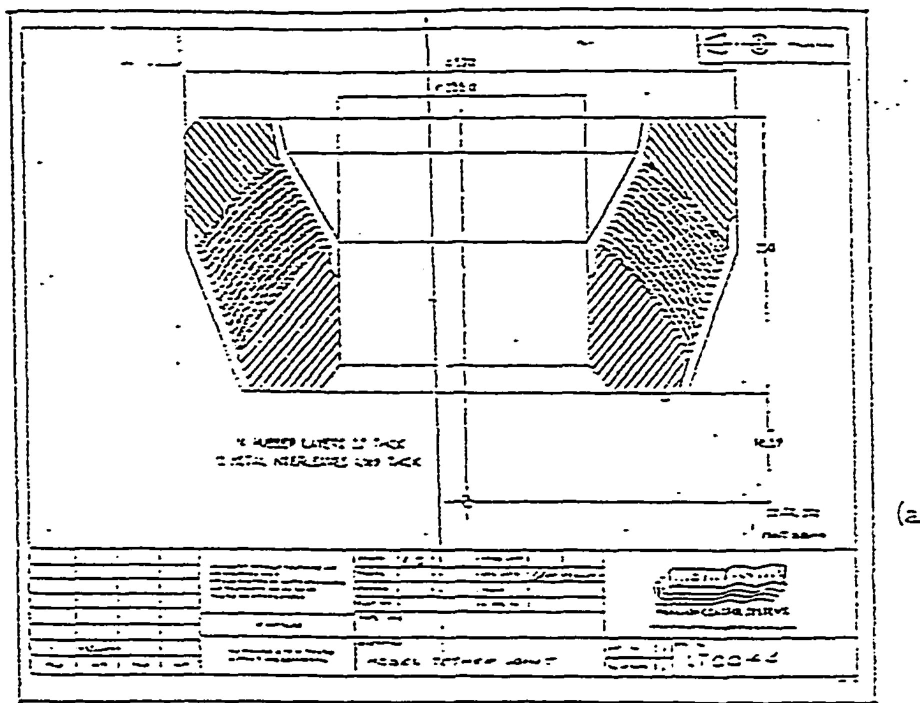


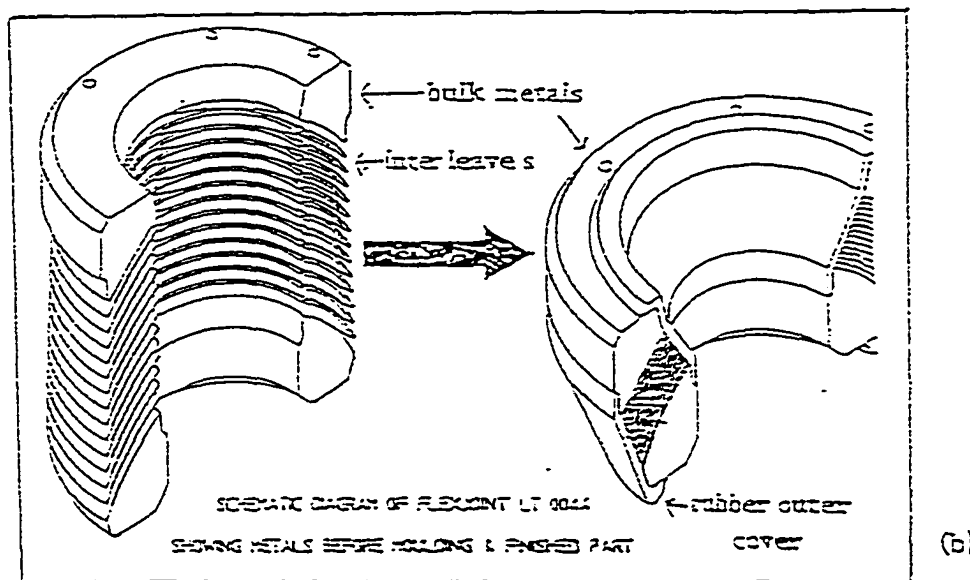
Fig. 7.4 A 3-D model of the TLP Flexjoint EXXON FEM

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION N° 2
TLP Flexjoint EXXON FEM



(a)



(b)

Fig. 7.5 Schematic diagram of the TLP Flexjoint EXXON FEM

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION N° 2

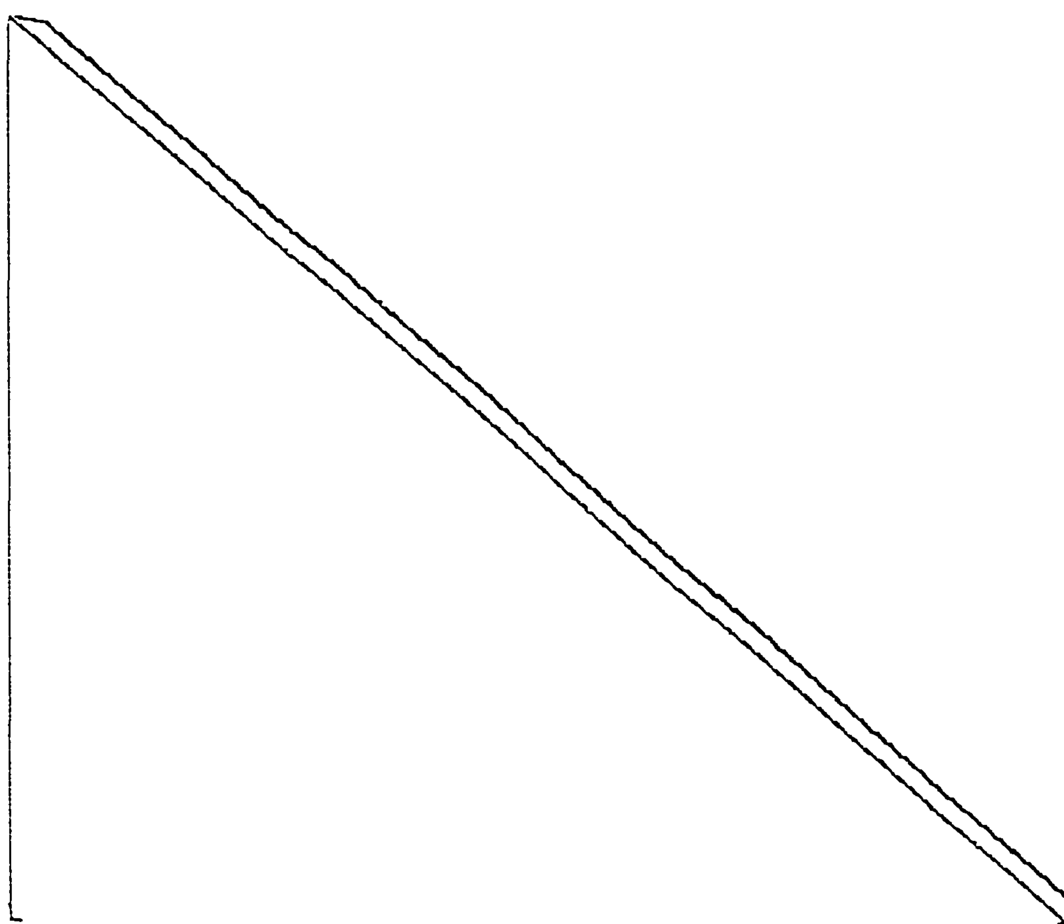


Fig. 7.6 Non-zero pattern of the TLP Flexjoint EXXON FEM

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION N° 2

Equations	Coefficients	Max Semi-bandwidth	Average Semi-bandwidth
23 ,155	809,427	689	665

Table 7.2 Characteristics of the TLP Flexjoint EXXON FEM

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION N° 3

Fig. 7.7 Non-zero pattern of the Thermal-Structural FEM

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION N° 3

Equations	Coefficients	Max Semi-bandwidth	Average Semi-bandwidth
43,806	1,037,705	31,956	1107

Table 7.3 Characteristics of the Thermal-Structural FEM

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

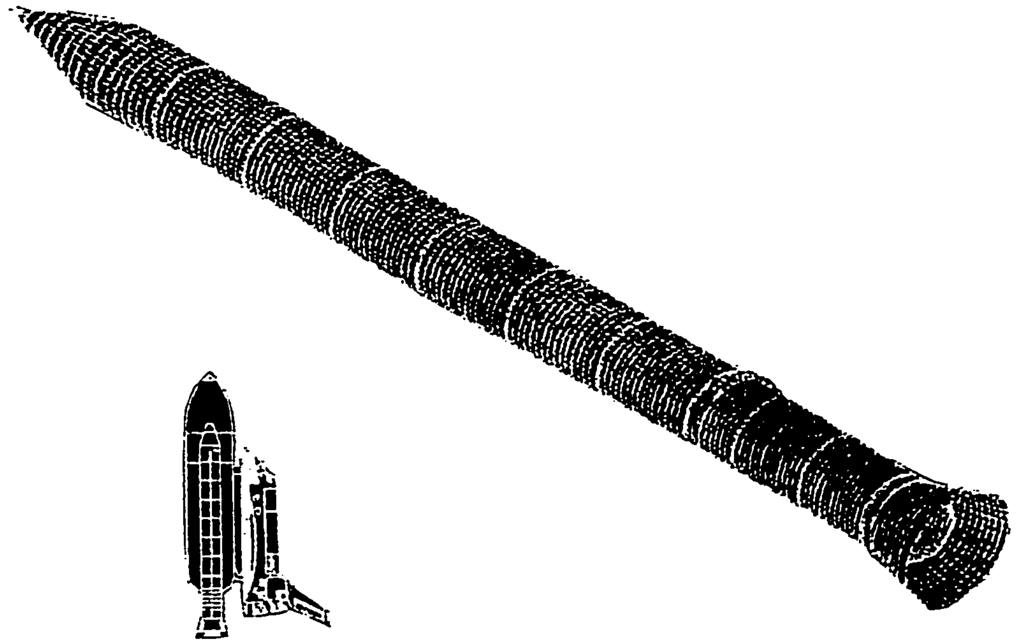


Fig. 7.8 FEM of the solid Rocket booster, SRB

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION N° 4

Equations	Coefficients	Max Semi-bandwidth	Average Semi-bandwidth
54,870	1,308,185	30,726	2,239

Table 7.4 Characteristics of the FEM Solid Rocket Booster, SRB

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION N° 4

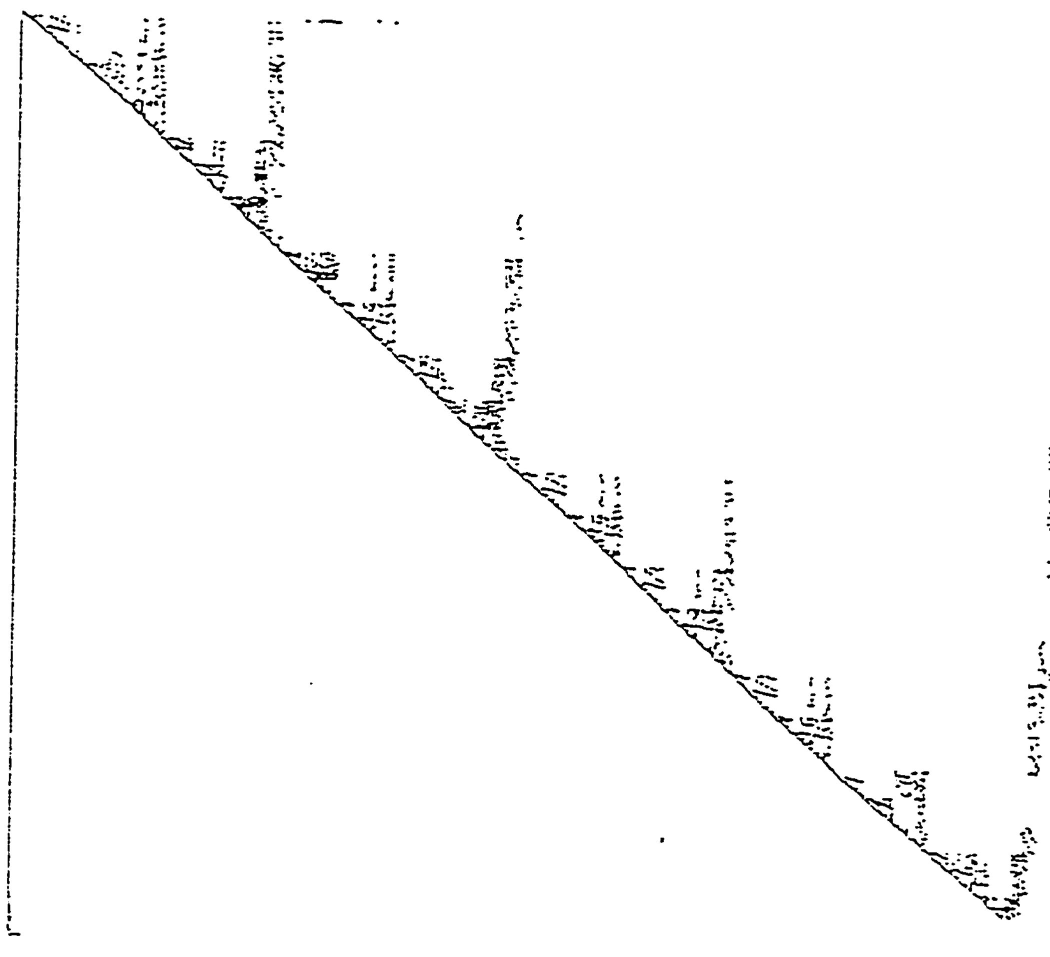


Fig. 7.9 Nonzero pattern of the FEM Solid rocket Booster, SRB

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION N° 5
Indefinite matrix: Cantiliver Beam Problem
51 DOF

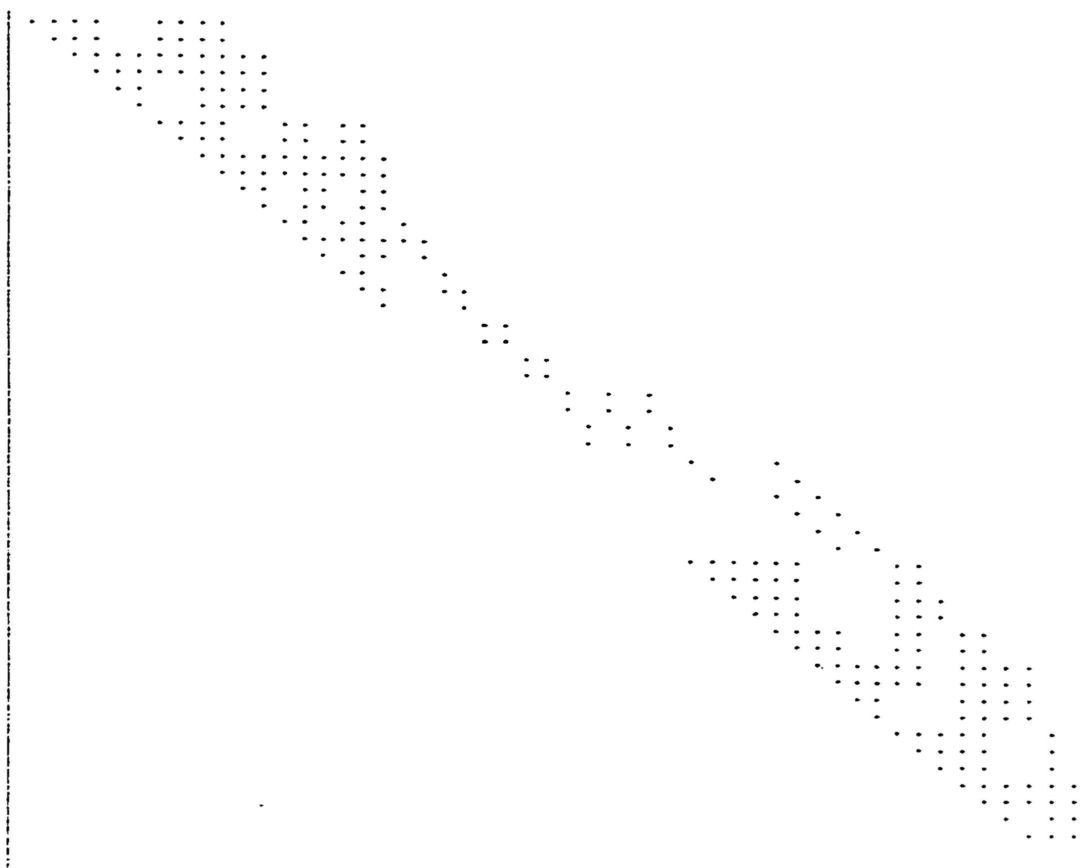


Fig. 7.10 Nonzero pattern of Application No 5
Cantilever Beam problem

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION N° 6
Indefinite matrix: Carlos Davilla Problem
247 DOF

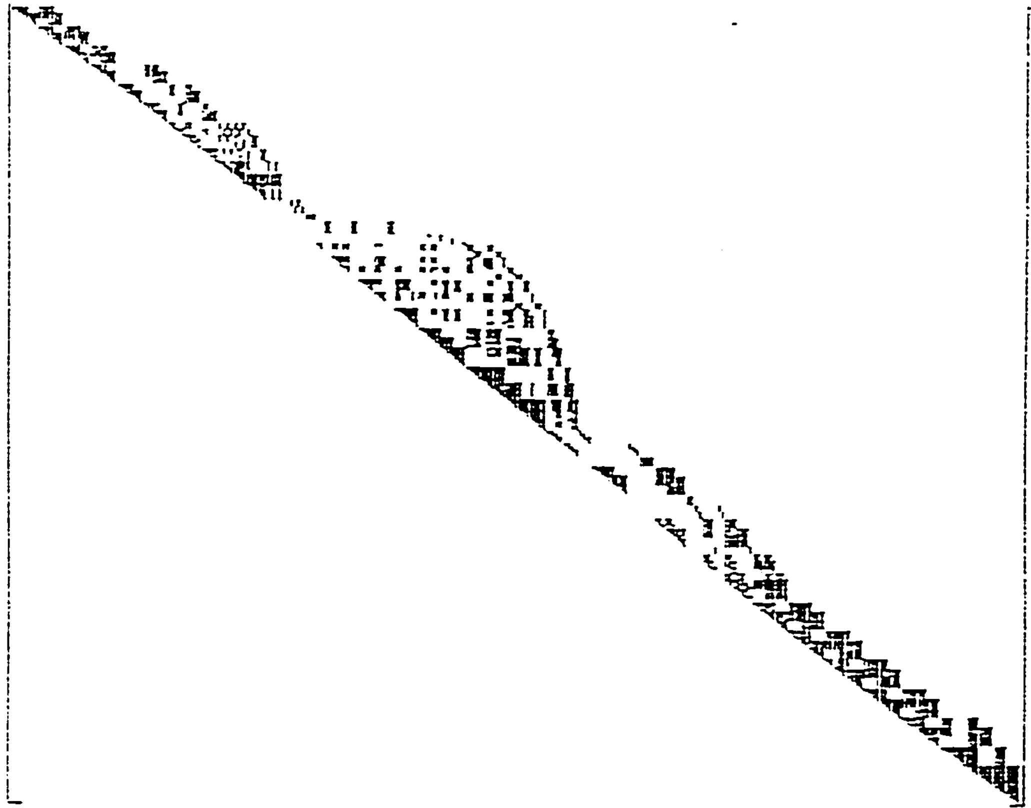


Fig. 7.11 Nonzero pattern of Application No 6
Carlos Davilla Problem

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION N° 7
Indefinite matrix: Jonathan's plate problem
1440 DOF

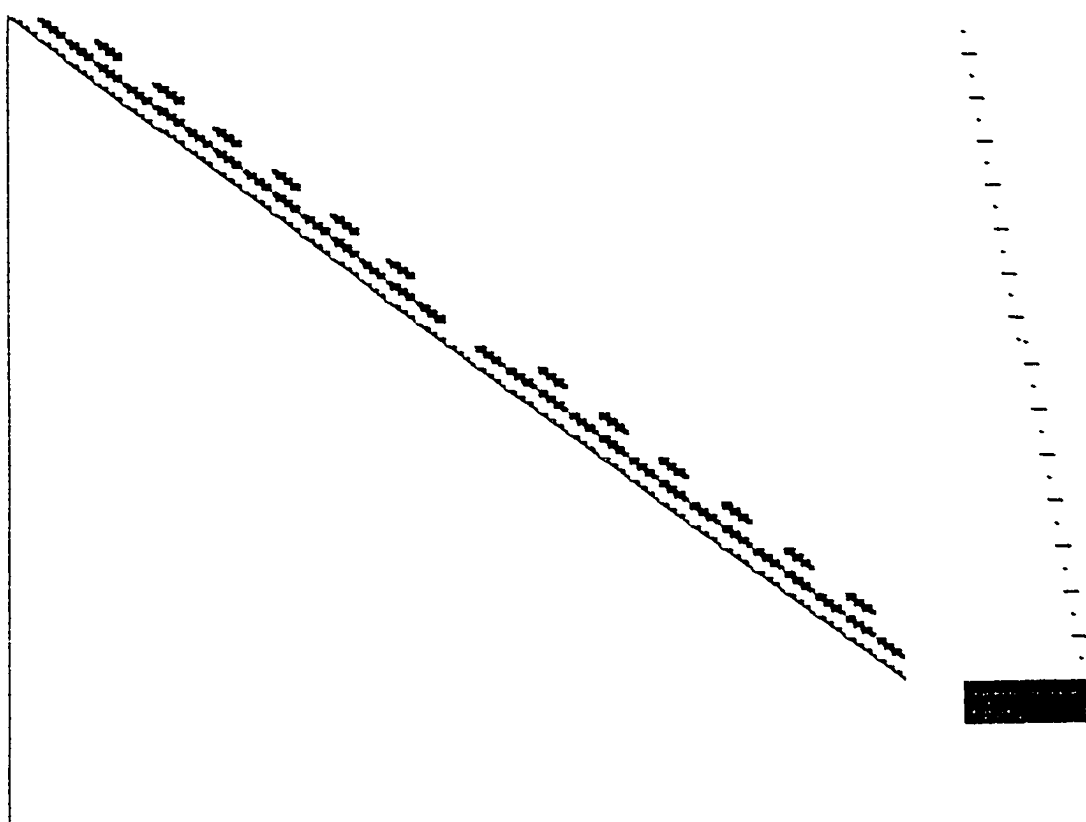


Fig. 7.12 Nonzero pattern of Application No 7
Jonathan's plate problem

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION N° 8
Indefinite matrix: Knight's Panel problem
2430 DOF

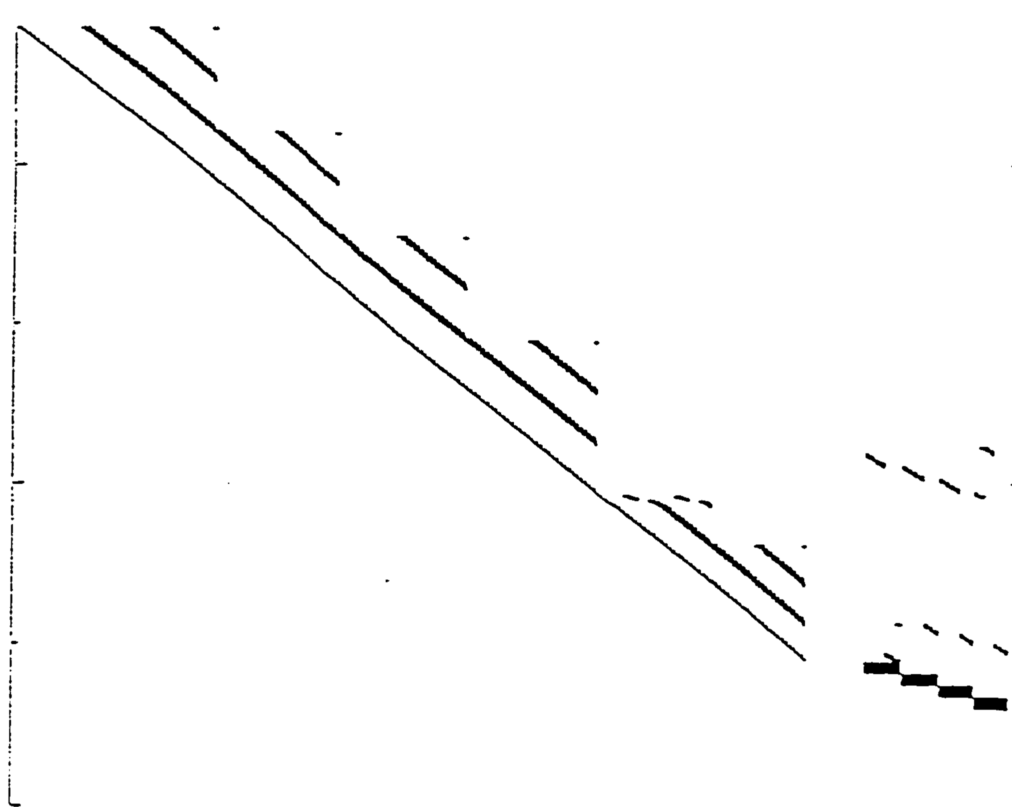


Fig. 7.13 Nonzero pattern of Application No 8
Knight's Panel problem

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION N° 9
Indefinite matrix: 15,367 problem

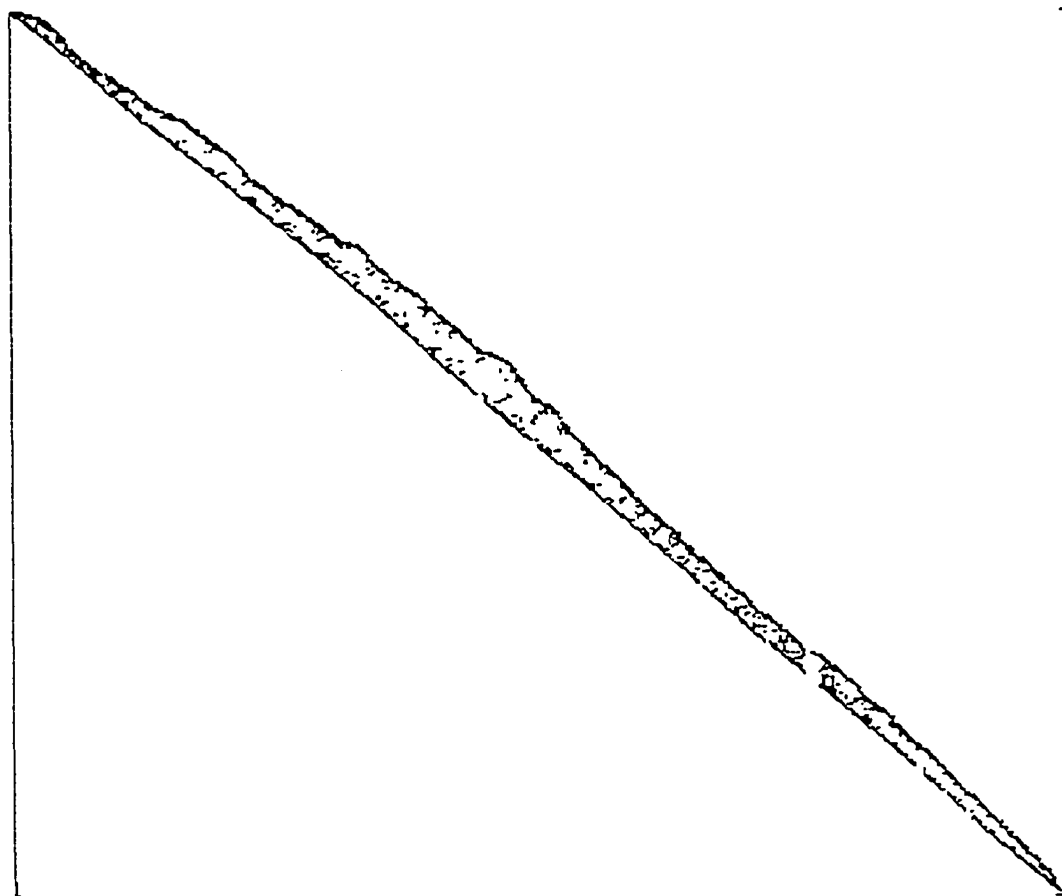


Fig. 7.14 Nonzero pattern of Application No 9
15,367 DOF indefinite problem

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Application	NEQ	NCOEF	Maximum semi-bandwidth	Average semi-bandwidth
No 5	51	218	11	7
No 6	247	2,009	44	17
No 7	1,440	22,137	1,246	143
No 8	2,430	75,206	1,100	280
No 9	15,367	286,044	1,035	514

Table 7.5 Characteristics of Indefinite matrices applications

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

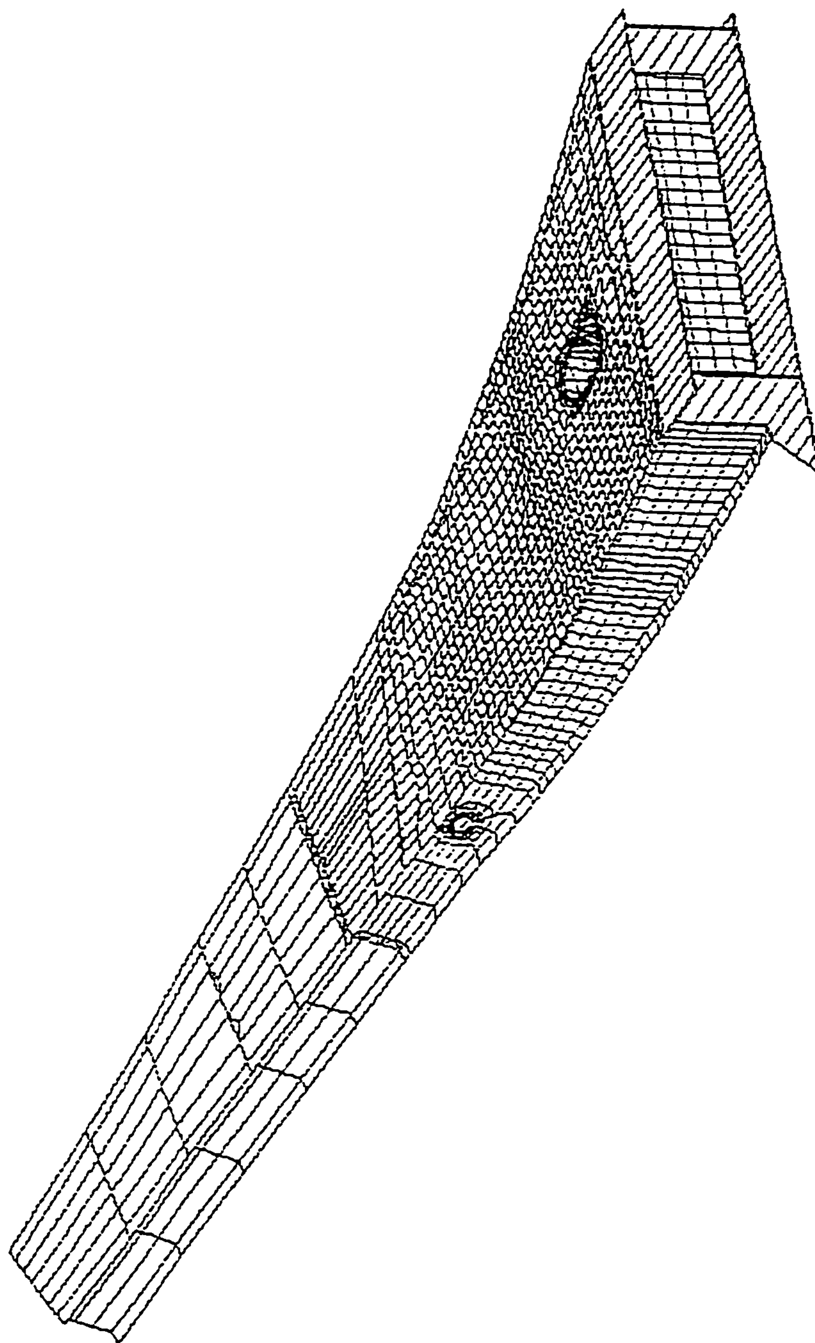


Fig. 7.15 Mcdonell Douglas Stiched/RFI All composite wing finite element model

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

REORD	NCOEF	NCOEF2	Integer Memory	Real Memory	Total Memory
No Reord	499,505	3,700,242	4,296,626	4,264,331	8,560,957
RCM	499,505	3,698,196	4,294,580	4,262,285	8,556,865
ND	499,505	3,210,738	3,807,122	3,774,827	7,581,949
MMD	499,505	3,017,283	3,613,667	3,581,372	7,195,039

Table 7.6 HSCT FEM: Memory requirement for different reordering algorithms

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

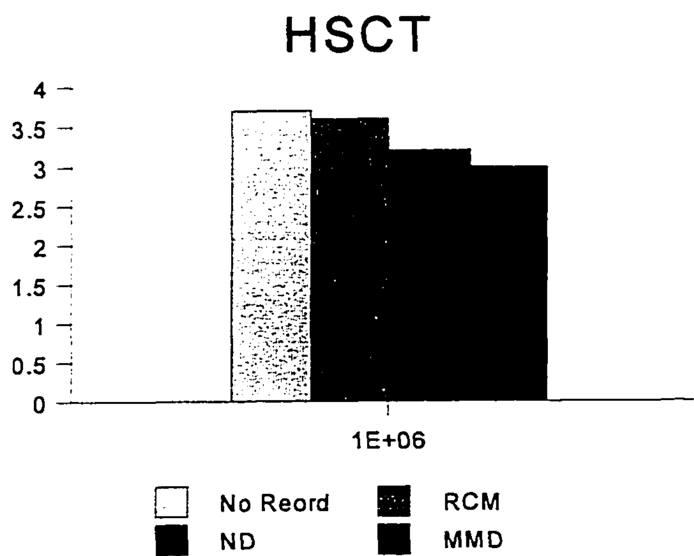


Fig. 7.16 HSCT FEM: Non-zeros elements after factorization for different reordering schemes

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LOOP unrol. Level	Symfa time (sec)	Numfa time (sec)	FBE time (sec)	Total time (sec)	Max abs displ	Sum abs displ	Relative Error Norm
1	0.480	31.630	0.300	34.910	0.447	301.291	1.34E-08
2	0.489	20.340	0.310	23.640	0.447	301.291	1.41E-08
8	0.480	16.880	0.310	20.160	0.447	301.291	1.36E-08

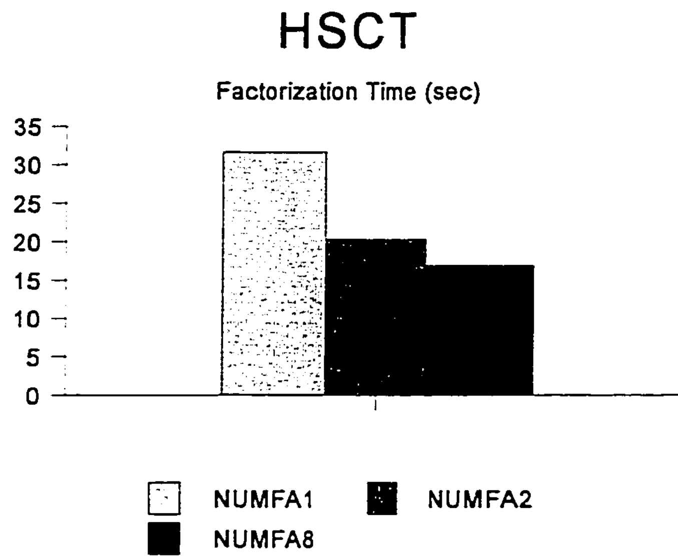
Table 7.7 HSCT FEM: Comparison of results using MMD and different level of loop unrolling on the IBM RS6000/590 *Stretch* machine.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LOOP unrol level	Symfa time (sec)	Numfa time (sec)	FBE time (sec)	Total time (sec)	Max abs displ	Sum abs displ	Relative Error Norm
1	3.921	505.437	5.315	539.501	0.447	301.291	1.41E-09
2	3.880	360.779	5.330	394.693	0.447	301.291	1.43E-09
8	3.881	247.448	5.311	281.274	0.447	301.291	1.43E-09

Table 7.8 HSCT FEM: Comparison of results using MMD and different level of loop unrolling on the Sun SPARC 20 *rhino* machine.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



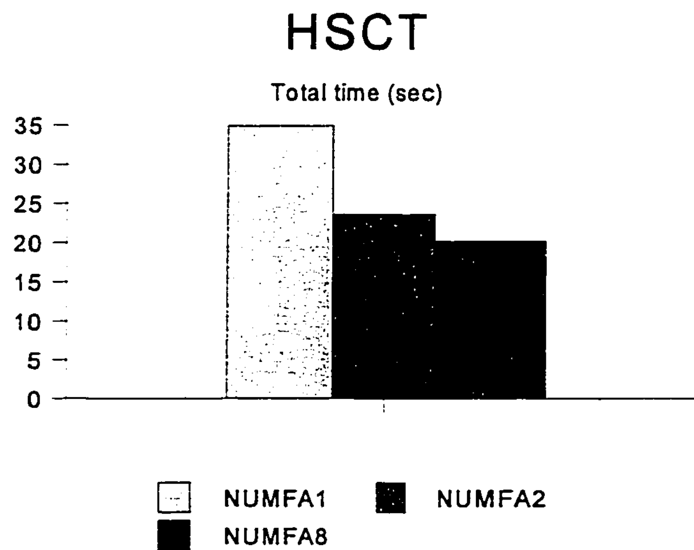
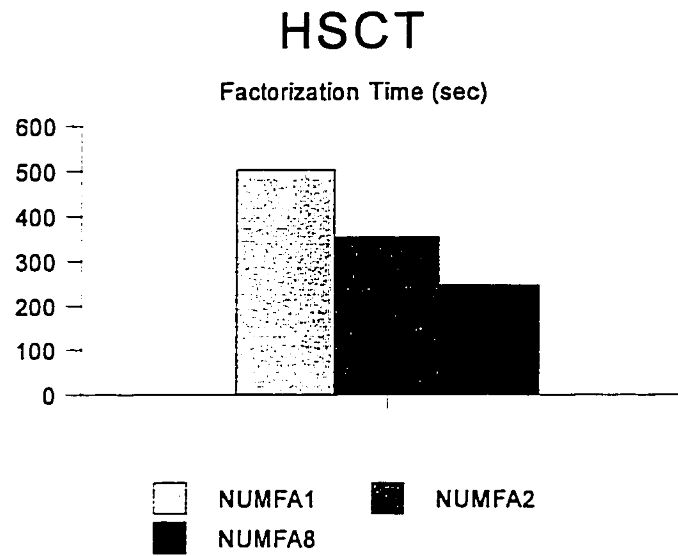


Fig. 7.17 HSCT FEM: Performance of Numfa1/2/8 on the *stretch* machine

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



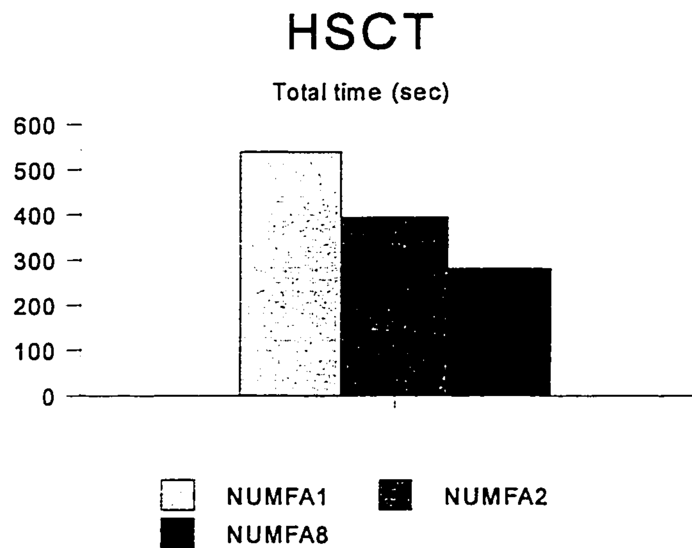
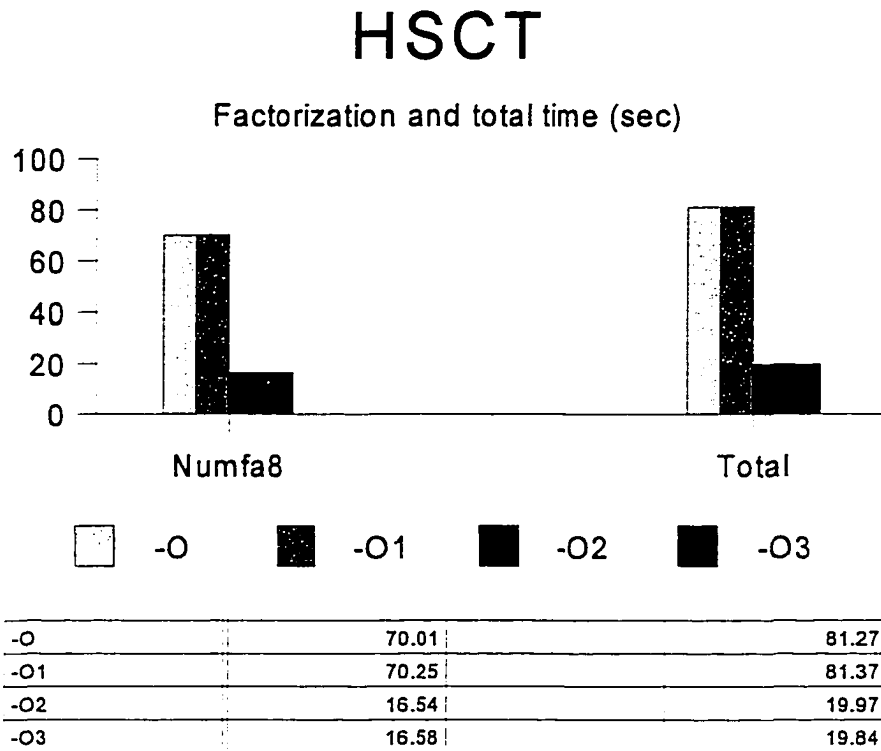


Fig. 7.18 HSCT FEM: Performance of Numfa1/2/8 on the *rhino* machine

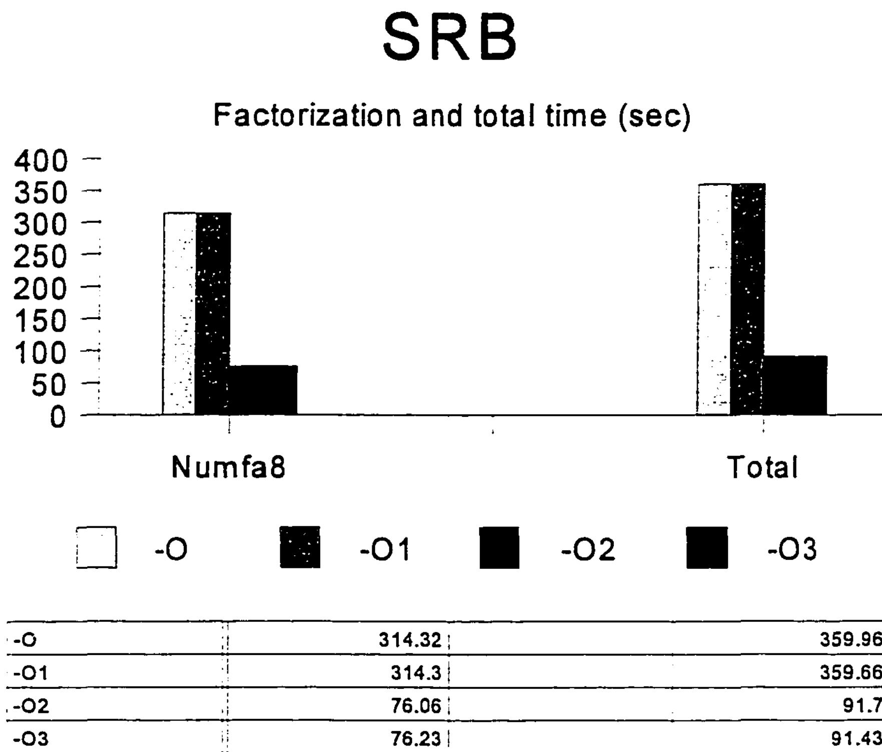
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



	Numfa8	Total
-O	70.01	81.27
-O1	70.25	81.37
-O2	16.54	19.97
-O3	16.58	19.84

Fig. 7.19 HSCT FEM: Performance of Numfa8 for different compiler optimization level on the *stretch* machine

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



	Numfa8	Total
-O	314.32	359.96
-O1	314.3	359.66
-O2	76.06	91.7
-O3	76.23	91.43

Fig. 7.20 SRB FEM: Performance of Numfa8 for different compiler optimization level on the *stretch* machine

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reord	Loop unrol level	Reord time (sec)	Symfa time (sec)	Numfa time (sec)	FBE time (sec)	Total time (sec)	Max abs displ	Sum abs displ	Relative Error Norm
No-Reord	1	-	0.700	32.040	0.380	35.120	0.447	301.291	0.22E-08
No-Reord	2	-	0.710	20.090	0.380	23.120	0.447	301.291	0.20E-08
No-Reord	8	-	0.690	16.040	0.380	19.110	0.447	301.291	0.21E-08
RCM	1	0.360	0.600	31.610	0.360	34.520	0.447	301.291	0.21E-08
RCM	2	0.350	0.600	20.190	0.380	23.060	0.447	301.291	0.21E-08
RCM	8	0.340	0.590	16.280	0.390	19.120	0.447	301.291	0.20E-08
ND	1	1.290	0.520	31.410	0.340	34.810	0.447	301.291	0.19E-08
ND	2	1.280	0.520	20.190	0.310	23.589	0.447	301.291	0.19E-08
ND	8	1.280	0.510	16.550	0.330	19.920	0.447	301.291	0.21E-08
MMD	1	0.254E-01	0.480	31.630	0.300	34.910	0.447	301.291	0.13E-08
MMD	2	0.261E-01	0.489	20.340	0.310	23.640	0.447	301.291	0.14E-08
MMD	8	0.261E-01	0.480	16.880	0.310	20.160	0.447	301.291	0.14E-08

Table 7.9 HSCT FEM: Summary of results on the IBM RS6000/590 *stretch* machine

Reord	Loop unrol level	Reord time (sec)	Symfa. time (sec)	Numfa time (sec)	FBE time (sec)	Total time (sec)	Max abs displ	Sum abs displ	Relative Error Norm
No-Reord	1	-	5.67	517.15	6.46	556.51	0.447	301.291	2.0E-09
No-Reord	2	-	5.66	368.00	6.45	407.34	0.447	301.291	2.1E-09
No-Reord	8	-	5.66	264.64	6.45	304.05	0.447	301.291	2.0E-09
RCM	1	2.94	4.87	517.10	6.49	555.63	0.447	301.291	1.9E-09
RCM	2	2.94	4.91	366.31	6.45	404.97	0.447	301.291	2.1E-09
RCM	8	2.94	4.87	267.78	6.47	306.36	0.447	301.291	2.0E-09
ND	1	11.15	4.15	496.88	5.633	532.51	0.447	301.291	2.0E-09
ND	2	11.12	4.14	348.02	5.63	383.61	0.447	301.291	1.9E-09
ND	8	11.12	4.15	242.55	5.66	278.17	0.447	301.291	1.9E-09
MMD	1	0.15	3.92	505.43	5.31	539.51	0.447	301.291	1.4E-09
MMD	2	0.15	3.88	360.78	5.32	394.69	0.447	301.291	1.4E-09
MMD	8	0.15	3.88	247.44	5.31	281.27	0.447	301.291	1.4E-09

Table 7.10 HSCT FEM: Summary of results on the Sun SPARC 20 *rhino* machine

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Loop unrol. Level	Symfa time (sec)	Numfa time (sec)	FBE time (sec)	Total time (sec)	Max abs displ	Sum abs displ.	Relative Error Norm
1	2.334	392.540	1.330	416.870	0.113E-04	0.561E-01	0.58E-10
2	2.239	241.010	1.300	265.620	0.113E-04	0.561E-01	0.59E-10
8	2.330	199.440	1.280	223.770	0.113E-04	0.561E-01	0.58E-10

Table 7.11 EXXON Off-shore FEM: Comparison of results using MMD and different level of loop unrolling on the IBM RS6000/590 *Stretch* machine.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reord	Loop unrol Level	Reord time (sec)	Symfa time (sec)	Numfa time (sec)	FBE time (sec)	Total time (sec)	Max abs displ	Sum abs displ	Relative Error Norm
MMD	1	0.41E-01	0.790	31.570	0.590	36.180	0.81E-12	0.81-12	0.18E-15
MMD	2	0.43E-01	0.780	20.380	0.590	24.860	0.81E-12	0.81-12	0.18E-15
MMD	8	0.42E-01	0.790	17.510	0.600	22.090	0.18E-12	0.18-12	0.18E-15

Table 7.12 Thermal-Structural FEM: Comparison of results using MMD and different level of loop unrolling on the IBM RS6000/590 *Stretch* machine.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reord	Loop unrol level	Reord time (sec)	Symfa time (sec)	Numfa time (sec)	FBE time (sec)	Total time (sec)	Max abs displ	Sum abs displ	Relative Error Norm
MMD	1	0.14E-01	2.080	146.90	1.290	161.670	2.061	13569.652	0.78E-12
MMD	2	0.14-E01	2.020	93.54	1.350	108.330	2.061	13569.652	0.81E-12
MMD	8	0.14E-01	2.240	76.85	1.350	92.450	2.061	13569.652	0.81E-12

Table 7.13 SRB FEM: Comparison of results using MMD and different level of loop unrolling on the IBM RS6000/590 *Stretch* machine.

HSCT AIRCRAFT MODEL
3, 8, 1 , 16146, 16146, 499505, 1, 0, 0, -1

nreord, loop, n3, neq, neq, ncoef, n7, n8, n9, mread

Table 7.14 HSCT FEM: K.INFO for Numfa8

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

HSCT AIRCRAFT MODEL

-TIME MMD = 0.2609395981E-01

OUTPUT SPARSE SOLVER

Number of Equations => NEQ = 16146
Non-Zero before fill in => NCOEF = 499505
Non-Zero after fill in => NCOEF2 = 3017283
Loop Unrolling Level => LOOP = 8

MEMORY

Total Integer memory = 3613667
Total real memory = 3581372
Total memory = 7195039

NORM CHECK

MAX ABS DISPL AT DOF 522 = 0.447440400042149411
SUMMATION OF ABS DISPLACEMENTS = 301.291343623234013
THE ABSOLUTE ERROR IS || Ax-b || = 0.192431628765362175E-06
THE RELATIVE ERROR IS || AX-b || / ||b|| = 0.136069709614759900E-08

TIMING

-TIME READ Fort.* files = 0.000000000000000000E+00
-TIME SYMFACT = 0.479999989271163940
-TIME TRANSA = 2.06999995373189449
-TIME SUPNODE Before N= 0.169999996200203896
-TIME NUMFA = 16.8799996227025986
-TIME FBE = 0.309999993070960045
-TIME SUPNODE After N = 0.169999996200203896
-TIME MULTSPA = 0.399999991059303284E-01
-TIME ERROR NORM = 0.000000000000000000E+00

Table 7.15 HSCT FEM: Output file of Numfa8 on the *stretch* machine

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Cache size (Kbytes)	Read K.* (sec)	Ordering (sec)	Symbolic Factorizat. (sec)	Numerical Factorizat. (sec)	Fbe (sec)	Maximum Absolute Displ	Summation Absolute Displ	Relative Error Norm
0.00	29.260	1.550	0.110	11.950	0.360	0.447	301.291	0.12E-08
32	29.960	1.650	0.120	10.920	0.350	0.447	301.291	0.12E-08
64	29.290	1.540	0.100	10.000	0.350	0.447	301.291	0.12E-08

Table 7.16 HSCT FEM: OakRigdeODU solver. Impact of cache size on the IBM RS6000/590 *stretch* machine using MMD

Loop Unrolling Level	Read K.* (sec)	Orderin g (sec)	Symbolic Factorizat. (sec)	Numerical Factorizat. (sec)	Fbe (sec)	Maximum Absolute Displ.	Summation Absolute Displ.	Relative Error norm
1	29.370	1.630	0.110	15.430	0.370	0.447	301.291	0.12E-08
2	28.760	1.590	0.110	10.680	0.370	0.447	301.291	0.12E-08
4	28.820	1.610	0.110	9.690	0.350	0.447	301.291	0.12E-08
8	29.290	1.540	0.100	10.000	0.350	0.447	301.291	0.12E-08

Table 7.17 HSCT FEM: OakRigdeODU solver. Impact of loop unrolling level on the IBM RS6000/590 *stretch* machine using MMD and cache size 64

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Cache size (Kbytes)	Read K.* (sec)	Ordering (sec)	Symbolic Factorizat. (sec)	Numerical Factorizat. (sec)	Fbe (sec)	Maximum Absolute Displ	Summation Absolute Displ	Relative Error Norm
0.00	42.384	2.735	0.145	64.048	1.297	0.447	301.291	0.12E-08
32	41.043	2.674	0.144	55.691	1.294	0.447	301.291	0.12E-08
64	41.039	2.692	0.147	58.082	1.295	0.447	301.291	0.12E-08

Table 7.18 HSCT FEM: OakRigdeODU solver. Impact of cache size on the IBM RS6000/590 *rhino* machine using MMD and loop 8

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Loop Unrolling Level	Read K.* (sec)	Orderin g (sec)	Symbolic Factorizat. (sec)	Numerical Factorizat. (sec)	Fbe (sec)	Maximum Absolute Displ	Summation Absolute Displ	Relative Error norm
1	40.753	2.687	0.144	102.289	1.296	0.447	301.291	0.12E-08
2	41.246	2.680	0.144	62.963	1.294	0.447	301.291	0.12E-08
4	40.816	2.678	0.144	58.586	1.295	0.447	301.291	0.12E-08
8	41.039	2.692	0.147	58.082	0.350	0.447	301.291	0.12E-08

Table 7.19 HSCT FEM: OakRigdeODU solver. Impact of loop unrolling level on the Sun SPARC 20 *rhino* machine using MMD and cache size 64

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Cache size (Kbytes)	Read K.* (sec)	Orderin g (sec)	Symbolic Factorizat. (sec)	Numerical Factorizat. (sec)	Fbe (sec)	Maximum Absolute Displ	Summation Absolute Displ	Relative Error Norm
0.00	50.890	1.170	0.170	143.040	1.510	0.113-04	0.561-01	0.25E-10
32	51.000	1.180	0.170	153.400	1.460	0.113-04	0.561-01	0.25E-10
64	49.820	1.140	0.180	130.500	1.450	0.113-04	0.561-01	0.25E-10

Table 7.20 EXXON Off-shore FEM: OakRigdeODU solver. Impact of cache size on the IBM RS6000/590 *stretch* machine using MMD and loop 8

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Loop Unrolling Level	Read K.* (sec)	Ordering (sec)	Symbolic Factorizat. (sec)	Numerical Factorizat. (sec)	Fbe (sec)	Maximum Absolute Displ	Summation Absolute Displ	Relative Error Norm
1	49.070	1.200	0.170	195.960	1.520	0.113E-04	0.561E-01	0.26E-10
2	50.040	1.160	0.170	144.470	1.760	0.113E-04	0.561E-01	0.25E-10
4	50.010	1.150	0.160	130.600	1.470	0.113E-04	0.561E-01	0.25E-10
8	49.820	1.140	0.180	130.500	1.450	0.113E-04	0.561E-01	0.25E-10

Table 7.21 EXXON Off shore FEM: OakRigdeODU solver. Impact of loop unrolling level on the IBM RS6000/590 *stretch* machine using MMD and cache size 64

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Cache Size (Kbytes)	Read K.* (sec)	Ordering (sec)	Symbolic Factorizat. (sec)	Numerical Factorizat. (sec)	Fbe (sec)	Maximum Absolute Displ	Summation Abs Displ	Relative Error Norm
0.00	59.81	6.900	0.240	12.240	0.610	0.81E-12	0.81E-12	0.17E-20
32	59.76	6.860	0.260	11.070	0.620	0.81E-12	0.81E-12	0.17E-20
64	60.31	6.890	0.240	10.930	0.600	0.81E-12	0.81E-12	0.17E-20

Table 7.22 Thermal-Structural FEM: OakRigdeODU solver. Impact of cache size on the IBM RS6000/590 *stretch* machine using MMD and loop 8

Loop Unrolling Level	Read K.* (sec)	Ordering (sec)	Symbolic Factorizat. (sec)	Numerical Factorizat. (sec)	Fbe (sec)	Maximum Absolute Displ	Summation Absolute Displ	Relative Error Norm
1	59.91	6.860	0.240	16.210	0.600	0.81E-12	0.81E-12	0.17E-20
2	60.11	6.900	0.240	11.310	0.620	0.81E-12	0.81E-12	0.17E-20
4	60.14	6.900	0.230	10.550	0.610	0.81E-12	0.81E-12	0.17E-20
8	60.31	6.890	0.240	10.930	0.600	0.81E-12	0.81E-12	0.17E-20

Table 7.23 Thermal-Structural FEM: OakRigdeODU solver. Impact of loop unrolling level on the IBM RS6000/590 *stretch* machine using MMD and cache size 64

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Cache Size (Kbytes)	Read K.* (sec)	Ordering (sec)	Symbolic Factorizat. (sec)	Numerical Factorizat (sec)	Fbe (sec)	Maximum Absolute Displ.	Summation Absolute Displ.	Relative Error Norm
0.00	128.11	3.630	0.260	51.10	1.34	2.061	13569.65	0.41E-12
32	126.40	3.640	0.270	46.05	1.34	2.061	13569.65	0.41E-12
64	128.93	3.700	0.260	42.95	1.33	2.061	13569.65	0.41E-12

Table 7.24 SRB FEM: OakRigdeODU solver. Impact of cache size on the IBM RS6000/590 *stretch* machine using MMD and loop 8

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Loop Unrolling Level	Read K.* (sec)	Ordering (sec)	Symbolic Factorizat. (sec)	Numerical Factorizat. (sec)	Fbe (sec)	Maximum Absolute Displ	Summation Absolute Displ	Relative Error Norm
1	127.84	3.670	0.270	65.92	1.31	2.061	13569.65	0.41E-12
2	127.50	3.650	0.280	43.27	1.32	2.061	13569.65	0.41E-12
4	128.65	3.690	0.270	41.09	1.34	2.061	13569.65	0.41E-12
8	128.93	3.700	0.260	42.95	1.33	2.061	13569.65	0.41E-12

Table 7.25 SRB FEM: OakRigdeODU solver. Impact of loop unrolling level on the IBM RS6000/590 *stretch* machine using MMD and cache size 64

```
SRB PROBLEM
2        64    4  54870  54870  1308185  0   0  -1   -1
icase, cachsz, level, neq,      neq,   NCOEF, n7, n8, n9, mread
```

Table 7.26 SRB FEM: K.INFO input file for OakRidgeODU solver.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
TITLE: SRB PROBLEM
TIME FOR READING Original NASA K.* Files      = 128.9299927
TIME FOR READING DATA                         =   6.040
TIME FOR CONSTRUCT ADJANCY MATRIX             =    .450
TIME FOR OAK FORMAT ANZF                      =    .900

ORDERING OPTION: 2 - MULTIPLE MINIMUM DEGREE

CACHE SIZE (IN KBYTES): 64

LOOP UNROLLING LEVEL: 8

NUMBER OF EQUATIONS                           =   54870
NUMBER OF NONZEROS NCOFF                      = 1308185
NUMBER OF NONZEROS (INCLUDING DIAG.)          = 2671240
NUMBER OF NONZEROS (EXCLUDING DIAG.)          = 2616370

TIME FOR CREATING FULL REPRESENTATION         =    .270

------------------------------------

TIME FOR COPYING ADJACENCY STRUCT.            =    .310
TIME FOR ORDERING                             =   3.700

TIME FOR SYMBOLIC FACT. SETUP                 =   1.300
TIME FOR SYMBOLIC FACTORIZATION               =    .260

TIME FOR NUMERICAL INPUT                      =   1.790

TIME FOR FACTORIZATION INIT.                  =    .020
TIME FOR NUMERICAL FACTORIZATION              =  42.950

TIME FOR TRIANGULAR SOLUTIONS                 =   1.330
------------------------------------

MAX ABS DISPL AT DOF 47041                    = 2.061863884795100 52
SUMMATION OF ABS DISPLACEMENTS                = 13569.6516772657978
THE ABSOLUTE ERROR IS || Ax-b ||              = 0.310314607028814793E-05
THE RELATIVE ERROR IS || AX-b || / ||b||      = 0.409059792384211927E-12
------------------------------------

TIME FOR COMPUTING ERROR                      =   6.100

STATISTICS

NUMBER OF SUPERNODES                          = 8033
NUMBER OF NONZEROS IN L                       = 12240705
NUMBER OF SUBSCRIPTS IN L                     = 218982
LARGEST SUPERNODE BY COLUMNS                  = 738
LARGEST SUPERNODE BY NONZEROS                 = 1377
SIZE OF TEMPORARY WORK STORAGE                = 202566
FACTORIZATION OPERATION COUNT                 = 4.8815177570D+09
TRIANGULAR SOLN OPERATION COUNT               = 4.8853080000D+07

NORMAL TERMINATION
```

Table 7.27 SRB FEM: OakRidgeODU solver. Output file on the *stretch* machine

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Application	NEQ	Number of Zeros on the Diagonal	Percentage of Zeros on the Diagonal
No 5	51	14	27.45
No 6	247	37	14.98
No 7	1440	240	16.67
No 8	2430	480	19.75
No 9	15367	1995	12.98

Table 7.28 Percentage of Zero diagonal values of the Indefinite matrices

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION	NEQ	NCOEF	NCOEF2	CPU(seconds)	R.E.N
No 5	51	218	373	0.000	1.4×10^{-15}
No 6	247	2,009	3,221	0.009	6.2×10^{-12}
No 7	1,440	22,137	51,573	0.299	1.2×10^{-12}
No 8	2,430	75,206	299,188	8.389	1.2×10^{-09}
No 9	15,367	286,044	2,884,093	76.809	9.6×10^{-11}

Table 7.29 ODU-HKUST indefinite solver: Summary of results on Rhino

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION	NEQ	NCOEF		Sum displ	Max displ.	CPU (secs)	Relative Error Norm
No 5	51	218	Boeing	2.265E-02	1.999E-03	0.041	7.00E-14
			ODU	2.265E-02	1.999E-03	0.003	1.45E-13
No 6	247	2009	Boeing	3.160	0.152	0.245	4.03E-10
			ODU	3.160	0.152	0.021	9.27E-10
No 7	1440	22137	Boeing	29.685	0.203	2.352	3.26E-10
			ODU	29.685	0.203	0.571	6.16E-10
No 8	2430	75206	Boeing	34.703	9.312E-02	7.736	9.97E-11
			ODU	34.680	9.311E-02	6.136	1.01E-11
No 9	15367	286044	Boeing	512.35	0.206	35.77	4.38E-11
			ODU	512.35	0.206	36.625	2.73E-09

Table 7.30 ODU-HKUST indefinite solver: Comparison of results on the Cray Y-MP

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION	NEQ NCOEF		Sum displ	Max displ.	CPU (secs)	Relative Error Norm
No 5	51 218	MMD	2.265E-02	1.999E-03	0.000	1.66E-15
		MMD-ZE	2.265E-02	1.999E-03	0.000	1.49E-15
No 6	247 2009	MMD	3.160	0.152	1.00E-02	9.76E-12
		MMD-ZE	3.160	0.152	1.00E-02	6.28E-12
No 7	1440 22137	MMD	29.685	0.203	0.510	6.81E-12
		MMD-ZE	29.685	0.203	0.300	1.25E-09
No 8	2430 75206	MMD	34.661	9.312E-02	7.000	2.82E-09
		MMD-ZE	34.702	9.311E-02	8.389	1.21E-09
No 9	15367 286044	MMD	512.35	0.206	181.029	9.37E-11
		MMD-ZE	512.35	0.206	76.809	9.59E-11

Table 7.31 ODU-HKUST indefinite solver: Impact of using MMD and Zero-End on the *Stretch* machine

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

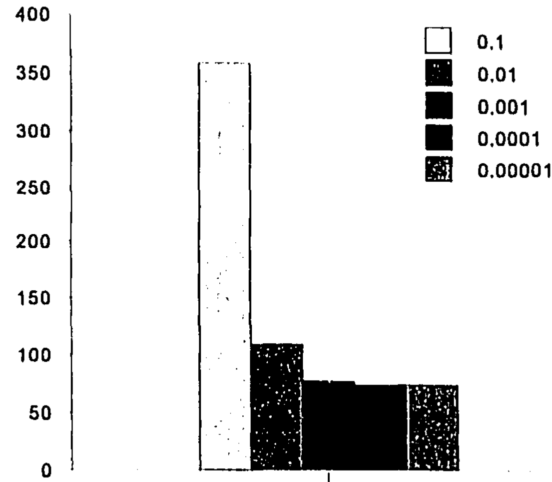


Fig. 7.21 ODU-HKUST indefinite solver: Impact of the control parameter alpha on application No 9 (neq =15367)

alpha	SUM	Max Displ	CPU(sec)	REN	#2x2 pivot	# diag inter	NCOEF2
0.1	512.355	0.206	358.499	1.69E-12	52	164	3632010
0.01	512.355	0.206	109.620	5.71E-10	10	45	2887346
0.001	512.355	0.206	76.810	2.59E-11	8	35	2884093
0.0001	512.350	0.206	73.399	8.64E-08	8	31	2883707
0.00001	517.757	0.206	73.509	6.10E-02	8	25	2883637

Table 7.32 ODU-HKUST indefinite solver: Impact of the control paramater alpha on application No 9 (neq =15367)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

NEQ	Max. abs.	Sum. abs.	Max. abs. displ. only	Sum. abs. displ. only	Numerical Fact. (sec)	Fbe (sec)	Total time (sec)	R.E.N
51	1.999	2.995	1.999	2.265	1.22E-02	1.28E-03	4.18E-02	1.7E-15
247	0.152	3.225	0.152	3.160	9.16E-02	8.35E-03	0.306	7.5E-12
1440	0.627	52.468	0.203	29.685	2.175	0.112	4.462	5.0E-12
2430	343849.445	6019377.075	9.31E-02	34.70	20.163	0.469	27.828	1.0E-13
15367	719.472	8472.301	0.206	512.354	307.008	3.918	339.102	9.3E-13

Table 7.33 ODU-Ma27: Summary of results on *Rhino* machine

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

NEQ	Max abs.	Sum. abs.	Max. abs. displ. only	Sum. abs. displ. only	Numerical Fact. (Sec)	Fbe (sec)	Total time (sec)	R.E.N.
51	1.999	2.995	1.999	2.265	0	0	0.199 E-01	0.260E-14
247	0.152	3.225	0.152	3.160	0.999E-02	0	0.150	0.681E-11
1440	0.627	52.468	0.203	29.685	0.140	0.999E-02	1.710	0.586E-11
2430	343849.445	6019377.075	9.31E-02	34.700	0.970	0.200E-01	6.170	0.111E-12
15367	719.472	8472.301	0.206	512.354	14.070	0.170	34.750	0.107E-11

Table 7.34 ODU-Ma27: Summary of results on *stretch* machine

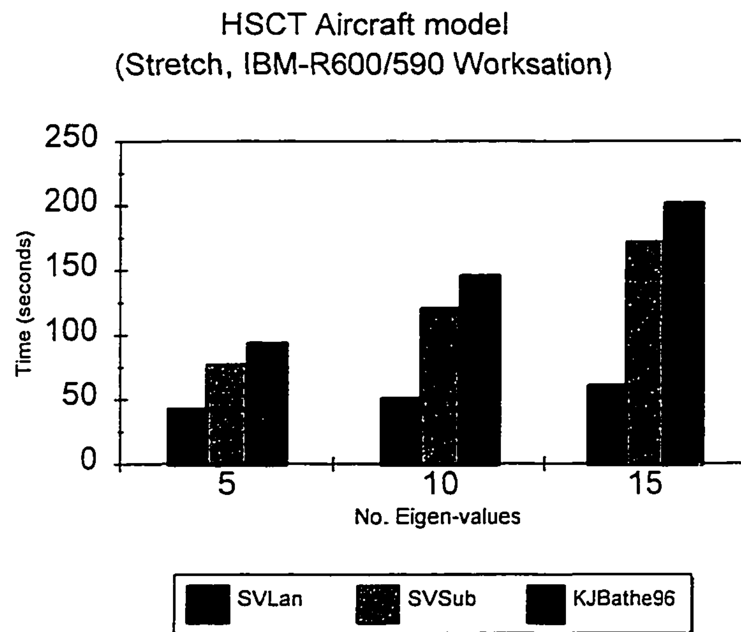


Fig. 7.22 HSCT FEM: Comparison of results for SPARSEPACK eigensolvers on *stretch*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

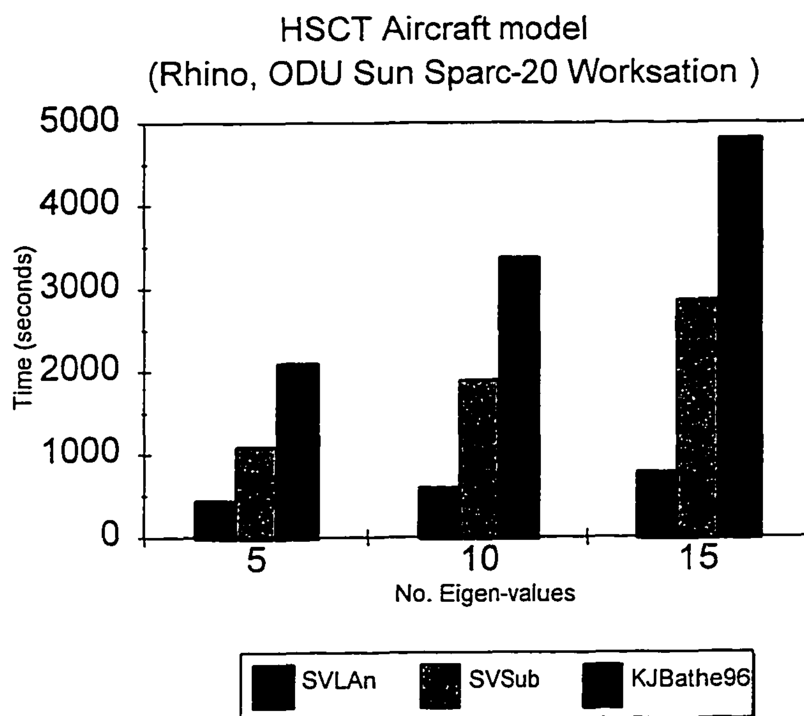


Fig. 7.23 HSCT FEM: Comparison of results for SPARSEPACK eigensolvers on *Rhino*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

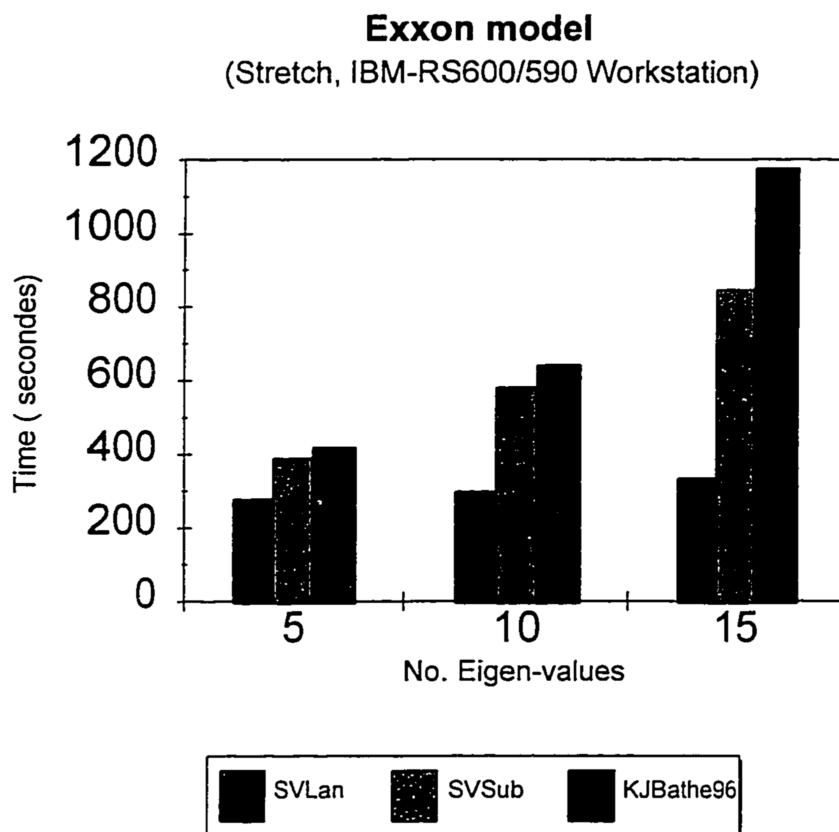


Fig. 7.24 EXXON Off-shore FEM: Comparison of results for SPARSEPACK eigensolvers on *stretch*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

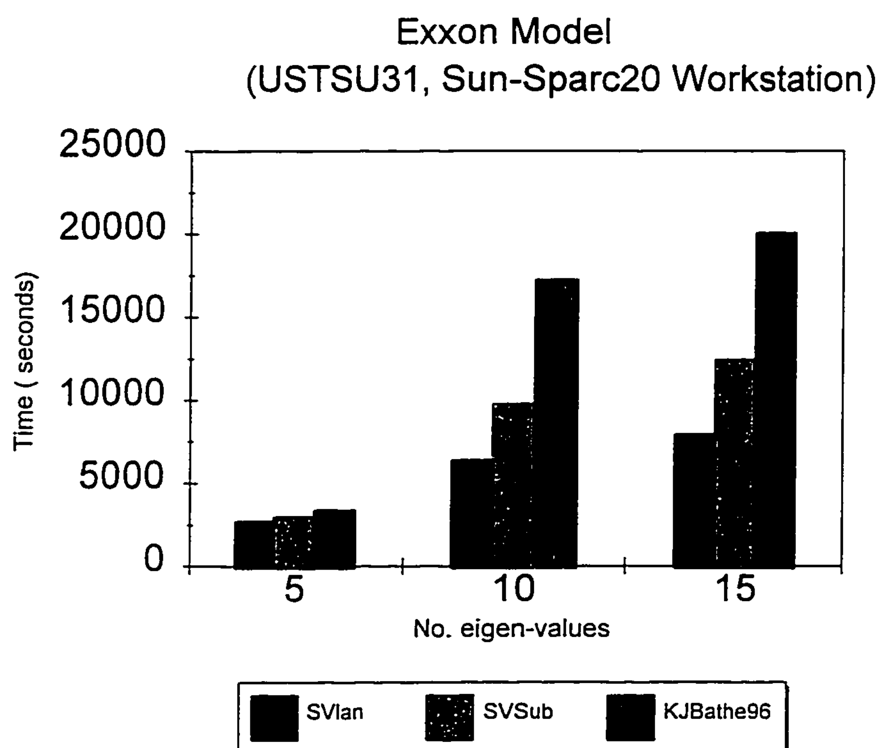


Fig. 7.25 EXXON Off-shore FEM: Comparison of results of SPARSEPACK eigensolvers on *USTSU31*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

HSCT Aircraft									
3	10	1	16,146	16,146	499,505	2	0	-1	-1
nord,	neig,	lump,	n,	n,	ncoeff,	x,	ishift,	iblock,	mread

Table 7.35 HSCT FEM: K.INFO for SPARSEPACK eigensolver

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
* OUTPUT VECTOR SPARSE LANCZOS *

CPU to get MD reordering = 0.6195330620E-01
neq                      = 23155
before fill in, ncoff    = 809427
after  fill in, ncof2    = 12842889
Total integer memory used =  13791249
Total real    memory used        =  13744936
** K****** EIG********HERTZ ****** ERROR ***** NORM ***
   1   .8816553E+03  .4725737E+01  .2289714E-20  .2790040E-08
   2   .1987976E+04  .7096197E+01  .8410216E-20  .1071936E-07
   3   .3806255E+04  .9819040E+01  .7461907E-19  .5147621E-08
   4   .5864529E+04  .1218812E+02  .9789936E-19  .1909064E-08
   5   .7608574E+04  .1388263E+02  .4796375E-19  .5057227E-09
   6   .7881169E+04  .1412913E+02  .9447560E-19  .8386751E-09
   7   .1090668E+05  .1662135E+02  .1299679E-20  .1635004E-08
   8   .1135674E+05  .1696082E+02  .7143559E-19  .3659260E-09
   9   .1406071E+05  .1887225E+02  .2549869E-13  .2099432E-06
  10   .1425347E+05  .1900117E+02  .6088567E-13  .5553453E-06

***TOTAL CPU FOR EIGENSOLUTION      = 299.879993297159672
***(This time including norm check & I/O )***
MTOTI = 14439610    MTOTA = 21655415
```

Table 7.36: EXXON Off-shore FEM: “ Sparse” Lanczos Algorithm from SPARSEPACK on stretch

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
CPU to get MD reordering = 0.8984327316E-02
 neq              = 23155
 before fill in, ncoff = 809427
 after fill in, ncof2 = 12842889
 Total integer memory used = 13791249
 Total real   memory used = 13744936
 IBLOCK = -1
   * OUTPUT VECTOR-SPARSE SUBSPACE ITERATION   *
 NEQ = 23155
 NCOEF= 809427
 IQ = 18
  RESULTS FOR EIGENSOLUTION
 Number of iterations = 13

 TOLERANCE CHECK ON EIGENVALUES

*# ******* EIGV***************** TOLJ***********
   1  881.655277374476100  0.000000000000000000E+00
   2  1987.97576276251743  0.766308953059224021E-14
   3  3806.25467918120876  0.140978972579276982E-13
   4  5864.52881890857770  0.341184842923635138E-14
   5  7608.57373151541742  0.882650167412288680E-12
   6  7881.16872545743081  0.746644428695344813E-13
   7  10906.6791124137235  0.723639523184758736E-10
   8  11356.7377577019506  0.402404711557253559E-08
   9  14060.7106850863383  0.294112550324989127E-08
  10  14253.4661497386078  0.450678313092431284E-06

 Timing
 Time normcheck =  0.6346702576E-03
***** # **,*** EIGV *** , *** HERTZ ***, ** ERROR NORM **
   1    .8816553E+03  .4725737E+01  .2687843E-08
   2    .1987976E+04  .7096197E+01  .1456036E-08
   3    .3806255E+04  .9819040E+01  .9797936E-09
   4    .5864529E+04  .1218812E+02  .1094771E-08
   5    .7608574E+04  .1388263E+02  .1250671E-06
   6    .7881169E+04  .1412913E+02  .3743034E-07
   7    .1090668E+05  .1662135E+02  .1658382E-05
   8    .1135674E+05  .1696082E+02  .1463919E-04
   9    .1406071E+05  .1887225E+02  .1459706E-04
  10    .1425347E+05  .1900117E+02  .2106256E-03
***TOTAL CPU FOR EIGENSOLUTION      = 570.619987245649099
***(This time including norm check & I/O )***
MTOTI = 14439610    MTOTA = 21655415
```

Table 7.37: EXXON Off-shore FEM: " Sparse" Subspace Iteration from SPARSEPACK on *stretch*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
NEQ = 23155
NROOT = 10


DEGREES OF FREEDOM EXITED BY UNIT STARTING ITERATION VECTORS
  23155.  22516.    17.    159.     2.    158.  22840.  22542.  22867.
  23136.
  22813.   1491.   2130.  22894.  22569.  23109.  22786.


CONVERGENCE REACHED FOR RTOL  .1000E-05
RELATIVE TOLERANCE REACHED ON EIGENVALUES
 .1000E-11 .7828E-07 .1000E-11 .4654E-07 .7860E-07 .1000E-11 .9161E-07 .1000E-11 .3
  812E-06 .3011E-06 .9069E-06 .3340E-05
 .5847E-05 .4316E-02 .1748E-02 .1171E-02 .1918E-01 .1903E-01


THE CALCULATED EIGENVALUES ARE
 .88165527740348E+03  .19879757627894E+04  .38062546792150E+04  .58645288189713E+04  .
  76085737315525E+04  .78811687254790E+04
 .10906679112447E+05  .11356737755975E+05  .14060710682344E+05  .14253465441538E+05
Number of Iteration = 19

PRINT ERROR NORMS ON THE EIGENVALUES
 .21571596855195E-08  .12976291259135E-08  .87877827191921E-09  .91294172010989E-09  .
  38721941458344E-09  .74276789108943E-09
 .37247039503038E-08  .38673782841283E-07  .36920504218012E-06  .29409601699681E-06
time for 1996 K.J. Bathe subspace iteration= 640.909985674545169
```

Table 7.38: EXXON Off-shore FEM: Using Basic K.J. bathe's Subspace Iteration (KJBATHE96) on *stretch*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
METHOD = 1
NEQ      = 247
NCOEF    = 2009
NEIG      = 15
ISHIFT    = 0
MREAD    = -1
LUMP      = 1
 * OUTPUT VECTOR-SPARSE SUBSPACE ITERATION  *
NEQ      = 247
NCOEF  = 2009
IQ        = 23
RESULTS FOR EIGENSOLUTION
Number of iterations = 27
TOLERANCE CHECK ON EIGENVALUES
 * # *,      * EIGV * ,                   * TOLJ*
  1 -26.9037089291829368  0.000000000000000000E+00
  2 -20.3774168334131396  -0.697382540258992640E-15
  3 -4.35999233084511317  -0.122226603026336059E-14
  4 -1.59417268262606648  -0.167142197839641113E-14
  5 -0.467720895678490900  -0.830790461168345158E-15
  6 -0.343197785608355954  -0.161746822267102397E-15
  7 0.654668644375400954  0.169585489417228152E-15
  8   4.95786919427879447  0.537435570542110461E-15
  9   6.39006568111283180  0.138993629177446775E-15
 10   6.53935222653818382  0.271641101115702013E-15
 11   14.7750567574227798  0.360680206221441547E-15
 12   22.6547608473375170  0.940918432838286630E-15
 13   28.9647498775422356  0.233047273608763022E-14
 14   31.9792099249090249  0.499591873941804227E-12
 15   33.3768755613979806  0.553500314784671545E-14
***** # **,*** EIGV *** , *** HERTZ ***, ** ERROR NORM **
   1  -.2690371E+02  .8255173E+00  .3401786E-08
   2  -.2037742E+02  .7184469E+00  .4888496E-12
   3  -.4359992E+01  .3323250E+00  .1907106E-12
   4  -.1594173E+01  .2009499E+00  .6217446E-12
   5  -.4677209E+00  .1088463E+00  .5255656E-12
   6  -.3431978E+00  .9323787E-01  .3046399E-12
   7   .6546686E+00  .1287748E+00  .1987108E-10
   8   .4957869E+01  .3543787E+00  .1473136E-11
   9   .6390066E+01  .4023211E+00  .2354323E-11
  10   .6539352E+01  .4069935E+00  .1042401E-11
  11   .1477506E+02  .6117651E+00  .3792497E-12
  12   .2265476E+02  .7575300E+00  .4683212E-10
  13   .2896475E+02  .8565545E+00  .1034847E-09
  14   .3197921E+02  .9000238E+00  .2193338E-06
  15   .3337688E+02  .9194814E+00  .2379166E-07
Time Subspace Iter. = 0.769999982789158821
Time Normcheck     = 0.999999977648258209E-02
     TIMING
Reordering   Time                 =  0.000000000000000000E+00
Factorization Time                =  0.999999977648258209E-02
Subspace+Normchecking  Time =  0.779999982565641403
Total Time + Junk                 =  0.789999982342123985
```

Table 7.39 Application No 6: Subspace iteration for indefinite systems

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
SPARSE VECTOR LANCZOS

METHOD  = 1
NEQ     = 247
NCOEF   = 2009
NEIG    = 15
ISHIFT  = 0
MREAD   = -1
LUMP    = 1

*** K, * EIG*,*HERTZ *,* ERROR *,* NORM *** iam
 1  .6546686E+00  .1287748E+00  .2894972E-21  .3339575E-10
 2  .4957869E+01  .3543787E+00  .1921880E-20  .3009163E-11
 3  .6390066E+01  .4023211E+00  .3245256E-20  .3457124E-11
 4  .6539352E+01  .4069935E+00  .4768359E-19  .1220441E-10
 5  .1477506E+02  .6117651E+00  .3812197E-19  .8208553E-11
 6  .2265476E+02  .7575300E+00  .3731293E-18  .4815349E-11
 7  .2896475E+02  .8565545E+00  .1960320E-18  .9372604E-12
 8  .3197921E+02  .9000238E+00  .3002066E-18  .8568683E-12
 9  .3337688E+02  .9194814E+00  .8950083E-19  .3103452E-11
10  .3705413E+02  .9688096E+00  .1176460E-17  .1987879E-11
11  .4647382E+02  .1084986E+01  .1045959E-10  .2221759E-06
12  .4838121E+02  .1107028E+01  .8310640E-08  .2179520E-03
13  .4874698E+02  .1111204E+01  .5380266E-08  .9080362E-04
14  .4963882E+02  .1121323E+01  .4536706E-08  .8307301E-04
15  .5509166E+02  .1181308E+01  .6224062E-05  .8717437E-01

JACOBIR: Steps in IAM =  59 0
Time Normcheck          = 0.9999999776482582209E-02

TIMING

Reordering    Time              =  0.9999999776482582209E-02
Factorization Time              =  0.9999999776482582209E-02
Lanczos+Normchecking  Time      =  0.1299999997094273567
Total Time  + Junk              =  0.1499999996647238731
```

Table 7.40 Application No 6: Lanczos iteration for indefinite systems

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
Jonathan's new 900 DOF ill-conditionned problem
 1     25    1     900   900  11989  1     0  0    -1
 n1, neig, lump, neq,  n5, NCOEF, l2,  ishift, n9, mread
```

Table 7.41: K.INFO input file for the Lanczos and Subspace eigensolver for indefinite systems

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

SPARSE VECTOR LANCZOS

```
METHOD = 1
NEQ    = 51
NCOEF  = 218
NEIG   = 15
ISHIFT = 0
MREAD  = -1

*** K, * EIG*,*HERTZ *,* ERROR *,* NORM *** iam
 1  0.4059576E+01 0.3206716E+00 0.2058520E-36 0.1065941E-12
 2  0.2760599E+02 0.8362224E+00 0.1082130E-34 0.5080295E-13
 3  0.5082506E+02 0.1134643E+01 0.2979595E-80 0.2356342E-13
 4  0.1060554E+03 0.1639029E+01 0.1615789E-63 0.4518568E-14
 5  0.1754717E+03 0.2108258E+01 0.3683013E-34 0.1186698E-13
 6  0.1907798E+03 0.2198297E+01 0.4004801E-34 0.9578321E-14
 7  0.2322176E+03 0.2425312E+01 0.1518127E-33 0.1132765E-13
 8  0.2453832E+03 0.2493116E+01 0.7733515E-34 0.1131854E-13
 9  0.2841360E+03 0.2682769E+01 0.1870519E-39 0.1113027E-13
10  0.3267951E+03 0.2877120E+01 0.5060790E-35 0.1068744E-13
11  0.3480848E+03 0.2969359E+01 0.1697806E-34 0.1272876E-13
12  0.3934068E+03 0.3156756E+01 0.4653216E-31 0.8213718E-14
13  0.3998996E+03 0.3182699E+01 0.1051193E-30 0.1038064E-13
14  0.4860665E+03 0.3508876E+01 0.2564164E-27 0.5742984E-14
15  0.5313186E+03 0.3668577E+01 0.4014661E-24 0.7493163E-14

JACOBIR: Steps in IAM = 51 0
Time Normcheck        = 2.44019031524660-02

TIMING

Reordering Time              = 6.69451057910920-03
Factorization Time           = 1.30524933338170-02
Lanczos+Normchecking Time    = 0.804196536540999
Total Time + Junk            = 0.823943540453391
```

Table 7.42 Application No 5: Lanczos for Indefinite systems on *Cedar* machine

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

25 Eigenvalues written to testbed library

	Eigenvalue	Hertz
1	.11123254E+04	.53080626E+01
2	.38659522E+04	.98957424E+01
3	.96398255E+04	.15626247E+02
4	.10806540E+05	.16544874E+02
5	.11636218E+05	.17168251E+02
6	.13203150E+05	.18287693E+02
7	.26499652E+05	.25908377E+02
8	.31733289E+05	.28351606E+02
9	.39754143E+05	.31733013E+02
10	.44644193E+05	.33628120E+02
11	.59908797E+05	.38955196E+02
12	.64964877E+05	.40565742E+02
13	.70187484E+05	.42164791E+02
14	.79874717E+05	.44980553E+02
15	.80586904E+05	.45180637E+02
16	.10528766E+06	.51642693E+02
17	.10578620E+06	.51764809E+02
18	.12193633E+06	.55575920E+02
19	.13110733E+06	.57628006E+02
20	.14051857E+06	.59660511E+02
21	.17301432E+06	.66200485E+02
22	.17776141E+06	.67102524E+02
23	.17859391E+06	.67259468E+02
24	.19050763E+06	.69466644E+02
25	.21357271E+06	.73551750E+02

Table 7.43 Jonathan's ill-conditionned problem: NASA Langley test bed results

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

SPARSE VECTOR LANCZOS

METHOD = 1
NEQ = 900
NCOEF = 11989
NEIG = 25
ISHIFT = 0
MREAD = -1

*** K, *	EIG*,*	HERTZ *,*	ERROR *,*	NORM *** iam
1	0.1258413E+04	0.5645881E+01	0.1820169E-21	0.1543513E-09
2	0.5300908E+04	0.1158765E+02	0.1557695E-21	0.9905331E-10
3	0.1059806E+05	0.1638450E+02	0.8965632E-21	0.1273354E-10
4	0.1808826E+05	0.2140516E+02	0.8588379E-21	0.1714142E-09
5	0.1856144E+05	0.2168333E+02	0.5579659E-20	0.2157528E-10
6	0.3698570E+05	0.3060815E+02	0.2340790E-20	0.4961125E-11
7	0.4599465E+05	0.3413295E+02	0.4491059E-20	0.1862382E-10
8	0.4816943E+05	0.3493059E+02	0.1091066E-20	0.7530044E-11
9	0.5766985E+05	0.3822034E+02	0.3000156E-19	0.9210959E-11
10	0.7238344E+05	0.4281932E+02	0.1037248E-19	0.9590764E-11
11	0.8301885E+05	0.4585730E+02	0.1682737E-18	0.5598636E-11
12	0.1064558E+06	0.5192838E+02	0.4732456E-19	0.9332571E-11
13	0.1227433E+06	0.5575952E+02	0.1889136E-18	0.3416530E-11
14	0.1330266E+06	0.5804827E+02	0.1193742E-18	0.7375778E-11
15	0.1343789E+06	0.5834257E+02	0.2014252E-19	0.4623631E-11
16	0.1467268E+06	0.6096419E+02	0.1242511E-18	0.9715841E-11
17	0.1752531E+06	0.6662741E+02	0.3740897E-19	0.7568847E-11
18	0.1878109E+06	0.6897323E+02	0.2482942E-18	0.2193613E-11
19	0.2044265E+06	0.7195959E+02	0.3582778E-19	0.3183317E-11
20	0.2141506E+06	0.7365120E+02	0.3772230E-18	0.1160874E-10
21	0.2312231E+06	0.7653071E+02	0.1894919E-18	0.2284011E-11
22	0.2748608E+06	0.8344043E+02	0.3243874E-18	0.7184275E-11
23	0.2788627E+06	0.8404566E+02	0.1390866E-18	0.2516133E-11
24	0.2860432E+06	0.8512085E+02	0.1316509E-17	0.3089432E-11
25	0.3145049E+06	0.8925527E+02	0.1997630E-17	0.1093807E-11

JACOBIR: Steps in IAM = 99 0
Time Normcheck = 0.68920135498047

TIMING

Reordering Time = 0.41382575035095
Factorization Time = 3.1836757659912
Lanczos+Normchecking Time = 33.785816669464
Total Time + Junk = 37.383318185806

Table 7.44 Jonathan's ill-conditionned problem: Lanczos on *rhino* machine

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
METHOD    = 1
 NEQ      = 900
 NCOEF    = 11989
 NEIG     = 25
 ISHIFT   = 0
 MREAD    = -1
  * OUTPUT VECTOR-SPARSE SUBSPACE ITERATION  *
NEQ        = 900
NCOEF      = 11989
IQ         = 33
RESULTS FOR EIGENSOLUTION
------------------------

Number of iterations = 54

TOLERANCE CHECK ON EIGENVALUES
 * # * ,   * EIGV * ,          * TOLJ*
  1   1258.4129273924   5.4204864832655D-16
  2   5300.9084183279   1.0294401977914D-15
  3   10598.055539777   1.2014398091264D-15
  4   18088.257669334   2.4134853938483D-15
  5   18561.444650889   2.9399480015023D-15
  6   36985.703682245   9.8361757244006D-16
  7   45994.646978548   1.2655312028075D-15
  8   48169.434756755   1.2083941031777D-15
  9   57669.850081532   1.2616571057315D-15
 10   72383.442611459   2.0103928057800D-16
 11   83018.851054889   1.7528446905084D-15
 12   106455.79691275   1.3669443703750D-15
 13   122743.29947674   4.7422271652798D-16
 14   133026.56452339   2.1878209484706D-16
 15   134378.85484023   2.1658043217690D-16
 16   146726.79635165   3.9670777499950D-16
 17   175253.06379278   1.6606745597980D-16
 18   187810.91960340   6.1985385127109D-16
 19   204426.49436150   4.2710457684512D-16
 20   214150.64005341   2.8539743772841D-15
 21   231223.08793308   0.
 22   274860.81542046   4.2354280892622D-16
 23   278862.67577493   4.1746469477649D-16
 24   286043.21889382   2.0349253912946D-16
 25   314504.94603029   6.1908287361791D-13

** # **,*** EIGV *** , *** HERTZ ***, ** ERROR NORM **
  1  0.1258413E+04  0.5645881E+01  0.8811063E-10
  2  0.5300908E+04  0.1158765E+02  0.7378048E-11
  3  0.1059806E+05  0.1638450E+02  0.1548159E-10
  4  0.1808826E+05  0.2140516E+02  0.4316036E-10
  5  0.1856144E+05  0.2168333E+02  0.2304214E-11
```

Table 7.45 Johnathan's ill-conditionned problem : Subspace on *rhino*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

6	0.3698570E+05	0.3060815E+02	0.6354915E-11
7	0.4599465E+05	0.3413295E+02	0.1569814E-11
8	0.4816943E+05	0.3493059E+02	0.3042919E-11
9	0.5766985E+05	0.3822034E+02	0.4533898E-10
10	0.7238344E+05	0.4281932E+02	0.3916547E-10
11	0.8301885E+05	0.4585730E+02	0.4076776E-11
12	0.1064558E+06	0.5192838E+02	0.2159606E-10
13	0.1227433E+06	0.5575952E+02	0.4064932E-11
14	0.1330266E+06	0.5804827E+02	0.3232312E-10
15	0.1343789E+06	0.5834257E+02	0.1057564E-09
16	0.1467268E+06	0.6096419E+02	0.3409693E-10
17	0.1752531E+06	0.6662741E+02	0.4370750E-10
18	0.1878109E+06	0.6897323E+02	0.1281822E-10
19	0.2044265E+06	0.7195959E+02	0.2583808E-10
20	0.2141506E+06	0.7365120E+02	0.8124793E-10
21	0.2312231E+06	0.7653071E+02	0.3010349E-10
22	0.2748608E+06	0.8344043E+02	0.9934881E-10
23	0.2788627E+06	0.8404566E+02	0.7918807E-10
24	0.2860432E+06	0.8512085E+02	0.3098537E-08
25	0.3145049E+06	0.8925527E+02	0.2833434E-06

Time Subspace Iter. = 335.21290111542
Time Normcheck = 0.69137573242188

TIMING

Reordering Time	=	0.41172361373901
Factorization Time	=	3.2129995822906
Subspace+Normchecking Time	=	335.95602989197
Total Time + Junk	=	339.58075308800

Table 7.45 Johnathan's ill-conditionned problem : Subspace on *rhino* (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

NGST Satellite model Eigenproblem

```
NEQ    = 5156
NCOEF  = 88966
NEIG   = 100
ISHIFT = 100
MREAD  = -1
LUMP   = 1
IBLOCK = 0
NREORD = 3
ITIME  = 1

CPU to get MD reordering = 0.1187491417E-01
neq                      = 5156
before fill in, ncoff    = 88966
after fill in, ncof2     = 208337
Total integer memory used = 328242
Total real memory used    = 317927
```

*** K******	EIG*********	HERTZ *******	ERROR ******	NORM ***
1	-.1009969E-06	.5057946E-04	.0000000E+00	.2379811E-08
2	-.5065785E-07	.3582148E-04	.0000000E+00	.2671863E-08
3	-.2848805E-07	.2686281E-04	.0000000E+00	.1395660E-08
4	-.1200462E-08	.5514350E-05	.0000000E+00	.9110841E-09
5	.9863612E-09	.4998482E-05	.0000000E+00	.2335762E-08
6	.1716943E-07	.2085442E-04	.0000000E+00	.1210234E-08
7	.3578351E+01	.3010659E+00	.0000000E+00	.2417579E-08
8	.4049292E+01	.3202651E+00	.0000000E+00	.1585694E-08
9	.1006407E+02	.5049019E+00	.0000000E+00	.1778258E-08
10	.1090430E+02	.5255561E+00	.0000000E+00	.1383876E-08
11	.2388019E+02	.7777482E+00	.0000000E+00	.3742180E-08
12	.1031162E+03	.1616157E+01	.0000000E+00	.1225205E-07
13	.1036423E+03	.1620274E+01	.0000000E+00	.2699103E-08
14	.1589046E+03	.2006265E+01	.0000000E+00	.5160873E-08
15	.1601737E+03	.2014261E+01	.0000000E+00	.2099135E-08
16	.1608443E+03	.2018473E+01	.0000000E+00	.1737954E-08
17	.1643077E+03	.2040089E+01	.0000000E+00	.4803450E-08
18	.3109959E+03	.2806710E+01	.0000000E+00	.1254371E-08
19	.3121301E+03	.2811823E+01	.0000000E+00	.4160239E-08
20	.6639871E+03	.4101096E+01	.0000000E+00	.4213715E-09
21	.7839603E+03	.4456226E+01	.0000000E+00	.2420110E-08
22	.8250824E+03	.4571606E+01	.0000000E+00	.2623791E-08
23	.8255635E+03	.4572939E+01	.0000000E+00	.1203455E-08
24	.9518780E+03	.4910331E+01	.0000000E+00	.8483968E-09
25	.9518782E+03	.4910331E+01	.0000000E+00	.1372158E-08
26	.9524710E+03	.4911860E+01	.0000000E+00	.7613478E-09
27	.9524710E+03	.4911860E+01	.0000000E+00	.7591269E-08
28	.9525859E+03	.4912156E+01	.0000000E+00	.3589516E-08
29	.1195543E+04	.5503042E+01	.0000000E+00	.2973814E-09
30	.1220745E+04	.5560739E+01	.0000000E+00	.4522305E-08
31	.1228218E+04	.5577734E+01	.0000000E+00	.2477209E-08
32	.1980327E+04	.7082533E+01	.0000000E+00	.1454931E-08
33	.2027125E+04	.7165728E+01	.0000000E+00	.2281194E-08
34	.2364884E+04	.7739717E+01	.0000000E+00	.6115332E-09
35	.2441923E+04	.7864771E+01	.0000000E+00	.1791366E-08
36	.2523301E+04	.7994746E+01	.0000000E+00	.1606471E-08
37	.2628826E+04	.8160205E+01	.0000000E+00	.1244105E-08
38	.3025730E+04	.8754578E+01	.0000000E+00	.1845398E-08
39	.3029738E+04	.8760374E+01	.0000000E+00	.9486021E-09
40	.3177517E+04	.8971480E+01	.0000000E+00	.1663053E-08
41	.3405606E+04	.9287896E+01	.0000000E+00	.2506008E-08
42	.3447575E+04	.9344950E+01	.0000000E+00	.3568277E-09
43	.3636287E+04	.9597302E+01	.0000000E+00	.2377967E-09

Table 7.46 NGST Satellite model (5156 DOF eigenproblem): Lanczos on *stretch*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

44	.4063720E+04	.1014570E+02	.0000000E+00	.9083975E-09
45	.5099385E+04	.1136525E+02	.0000000E+00	.1624943E-08
46	.6070541E+04	.1240035E+02	.0000000E+00	.4374237E-08
47	.7390347E+04	.1368209E+02	.0000000E+00	.3569484E-09
48	.7394760E+04	.1368618E+02	.0000000E+00	.1091757E-08
49	.7429089E+04	.1371791E+02	.0000000E+00	.3927519E-09
50	.7747490E+04	.1400879E+02	.0000000E+00	.2772750E-08
51	.7810990E+04	.1406608E+02	.0000000E+00	.1178507E-08
52	.7986601E+04	.1422332E+02	.0000000E+00	.1283644E-08
53	.7988468E+04	.1422499E+02	.0000000E+00	.1624890E-09
54	.8039051E+04	.1426995E+02	.0000000E+00	.1470223E-08
55	.8039054E+04	.1426995E+02	.0000000E+00	.1864628E-08
56	.8045958E+04	.1427608E+02	.0000000E+00	.5250802E-08
57	.8045958E+04	.1427608E+02	.0000000E+00	.4650046E-08
58	.8047091E+04	.1427709E+02	.0000000E+00	.6579347E-08
59	.8439815E+04	.1462132E+02	.0000000E+00	.2916738E-08
60	.8493737E+04	.1466795E+02	.0000000E+00	.1858620E-08
61	.8733642E+04	.1487366E+02	.0000000E+00	.6337748E-08
62	.8962721E+04	.1506746E+02	.0000000E+00	.3534875E-08
63	.9848984E+04	.1579486E+02	.0000000E+00	.1414466E-08
64	.9905333E+04	.1583998E+02	.0000000E+00	.1520590E-08
65	.1106567E+05	.1674207E+02	.0000000E+00	.4710718E-08
66	.1111635E+05	.1678035E+02	.0000000E+00	.2901364E-08
67	.1115026E+05	.1680593E+02	.0000000E+00	.3805108E-08
68	.1162314E+05	.1715860E+02	.0000000E+00	.2658956E-08
69	.1166786E+05	.1719158E+02	.0000000E+00	.2026304E-09
70	.1681017E+05	.2063508E+02	.0000000E+00	.4184248E-08
71	.2231741E+05	.2377618E+02	.0000000E+00	.5376996E-09
72	.2238761E+05	.2381354E+02	.0000000E+00	.1701393E-08
73	.2404625E+05	.2467992E+02	.0000000E+00	.8327529E-08
74	.2460007E+05	.2496251E+02	.0000000E+00	.2035491E-08
75	.2489330E+05	.2511085E+02	.0000000E+00	.6802017E-09
76	.2498622E+05	.2515767E+02	.0000000E+00	.7689179E-08
77	.2539394E+05	.2536210E+02	.0000000E+00	.5231510E-08
78	.2542395E+05	.2537708E+02	.0000000E+00	.3469535E-08
79	.3136447E+05	.2818637E+02	.0000000E+00	.1545523E-08
80	.3156790E+05	.2827763E+02	.0000000E+00	.4138619E-09
81	.3165950E+05	.2831862E+02	.0000000E+00	.3554204E-09
82	.3168183E+05	.2832861E+02	.0000000E+00	.7477802E-08
83	.3203707E+05	.2848699E+02	.0000000E+00	.1363612E-07
84	.3203707E+05	.2848699E+02	.0000000E+00	.9968770E-08
85	.3204669E+05	.2849127E+02	.0000000E+00	.1637359E-08
86	.3226505E+05	.2858817E+02	.0000000E+00	.5232148E-10
87	.3259738E+05	.2873502E+02	.0000000E+00	.1067081E-08
88	.3393257E+05	.2931761E+02	.0000000E+00	.3433675E-08
89	.3395826E+05	.2932870E+02	.0000000E+00	.7391951E-10
90	.3589878E+05	.3015504E+02	.0000000E+00	.3478503E-08
91	.3711287E+05	.3066072E+02	.0000000E+00	.2985016E-08
92	.3804519E+05	.3104345E+02	.0000000E+00	.6213089E-09
93	.3919935E+05	.3151081E+02	.0000000E+00	.3662602E-08
94	.3935903E+05	.3157492E+02	.0000000E+00	.2352208E-08
95	.4177678E+05	.3253026E+02	.0000000E+00	.1760624E-08
96	.4190986E+05	.3258204E+02	.0000000E+00	.8170915E-08
97	.4270187E+05	.3288846E+02	.0000000E+00	.1843430E-09
98	.4548740E+05	.3394421E+02	.4329127-317	.2360722E-08
99	.4591367E+05	.3410289E+02	.1149076-313	.2640223E-08
100	.5020741E+05	.3566186E+02	.2167838-305	.3240370E-08

JACOBIR: Steps in IAM = 399 0
***TOTAL CPU FOR EIGENSOLUTION = 50.7799988649785519
(This time including norm check & I/O)
MTOTI = 14439610 MTOTA = 21655415

Table 7.46 NGST Satellite model (5156 DOF eigenproblem): Lanczos on *stretch* (Continued)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPLICATION	NEQ	NCOEF	COMMENTS
No 11	2	1	Feasible region exist
No 12	2	1	One point feasible region
No 13	2	1	No Feasible region
No 14	2	1	Multiple solution
No 15	2	1	Feasible region exist

Table 7.47: IPM: Small scale Examples (for validating purposes)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Application	NEQ (=N)	NCOEF input-data	NCOEF data plus surplus & lack variables	NCOEF $\hat{A}\hat{A}^T$	NCOEF2 After factorization $\hat{A}\hat{A}^T$	NCOEF2/ N(N+1)/2	comments	optimization time (seconds)
No 16	51	218	269	229	242	0.18	Indefinite	0.195
No 17	51	487	538	300	355	0.26	Definite	0.209
No 18	247	2009	2256	2806	3015	0.09	Indefinite	0.762
No 19	247	4265	4512	3165	3488	0.11	Indefinite	0.618
No 20	1440	22137	23577	31638	88798	0.08	Indefinite	19.67

Table 7.48 IPM: Medium-Scale Examples (for timing purposes) on *Cedar* Sun workstation

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

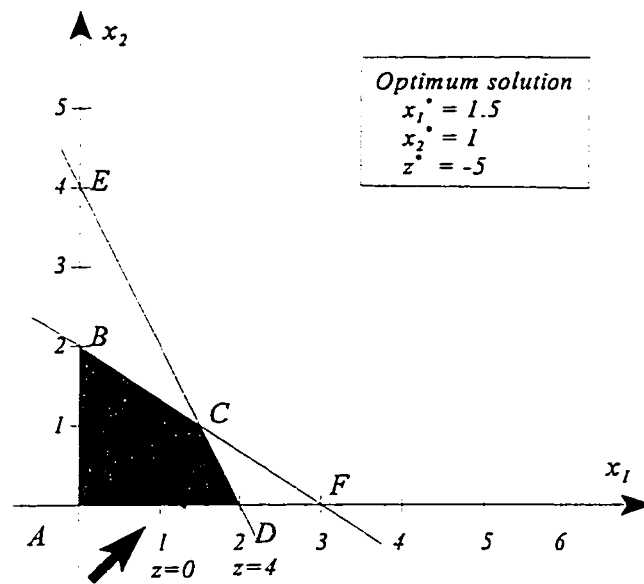


Fig. 7.26 IPM: Graphical solution application No 11

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

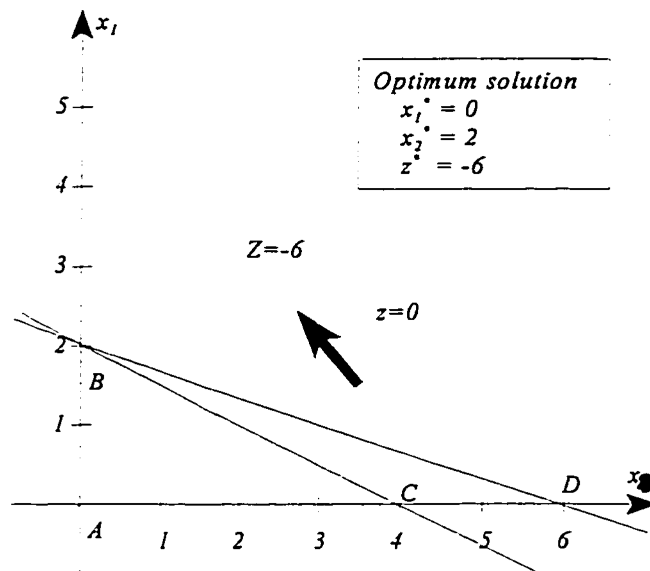


Fig. 7.27 IPM: Graphical solution application No12

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

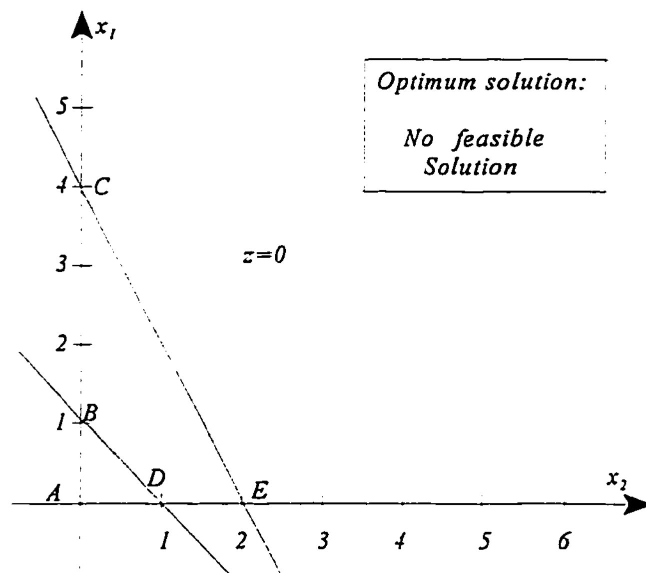


Fig. 7.28 IPM: Graphical solution application No 13

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

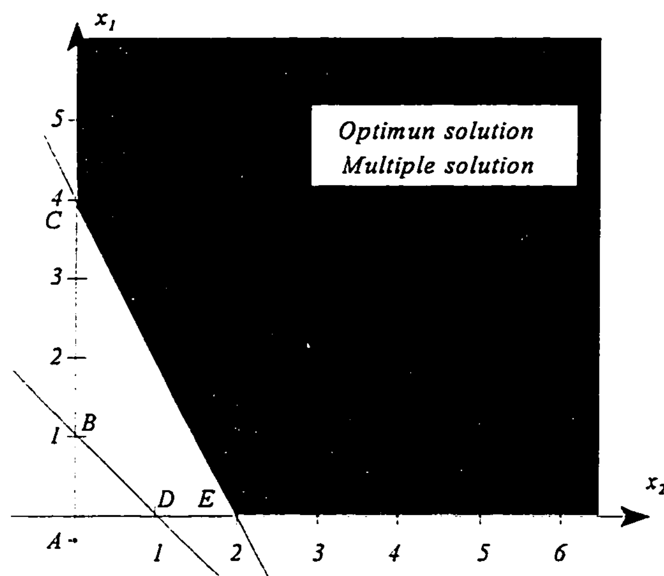


Fig. 7.29 IPM: Graphical solution application No 14

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

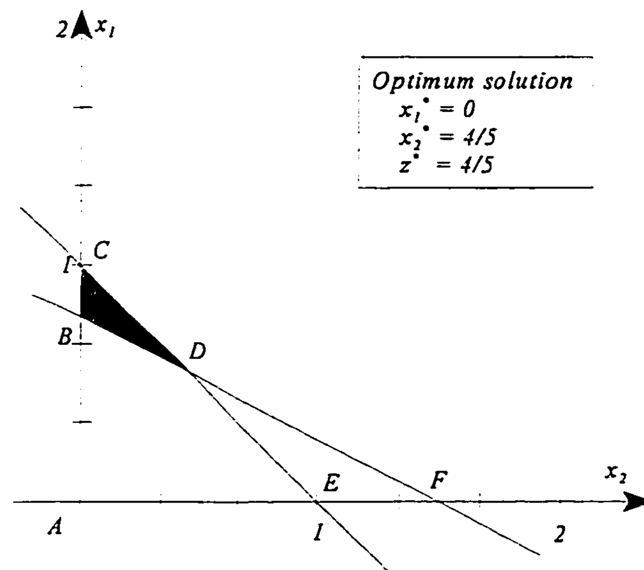


Fig. 7.30 IPM: Graphical solution application No 15

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

51equations:									
51	51	0	0	51	218	0	0	1	1
nv,	nl,	ng,	ne,	n,	NCOEF,	n7,	n8,	isolver,	mread

Table 7.49 K.INFO input file for the IPM

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
------------ OUTPUT INTERIOR METHOD ------------

51 DOF Problem

NV    = 51
NL    = 51
NG    = 0
NE    = 0
NC    = 102
NR    = 51
NCOEF = 218
NCOEF2 = 269
MREAD  = -1
ISOLVER= '

Time Read ORIGINAL DATA        =  4.6460501849651D-02
Time Define Basic set        =  8.5049867630005D-04
Time Construct Starting Vector =  3.5050511360168D-04

NUMBER OF ITERATION = 6

sparsity of [A]       = 269  5202   5.1710880430604D-02
sparsity of [A^ ]*[A^ ]"= 509  2601   0.19569396386005

Time Optimizer Phase II       =  0.19156400859356

OPTIMUM DESIGN POINT
  3.7242644508631D-06  3.7695123215782D-06  4.5281221399649D-06
  1.2049950134656D-06  9.2653967507183D-07  5.1596183718342D-06
  4.9547387683292D-06  1.2110670847008D-06  2.8317110676706D-06
  7.0698231129301D-07  4.6902771696144D-07  1.9168673384084D-06
  5.2374781958341D-06  3.8133009495046D-06  5.1779540534707D-06
  1.7816556372331D-06  4.1136587781992D-07  2.6473167128877D-06
  2.0125691617394D-06  1.9741306345507D-06  3.6080360085376D-06
  4.9716918361115D-06  5.1397870219177D-06  5.1397870219177D-06
  5.1397870219177D-06  5.1397870219177D-06  4.5538744344257D-06
  4.5538744344257D-06  4.5538744344257D-06  4.5538744344257D-06
  4.5538744344257D-06  4.5538744344257D-06  4.4361795026054D-06
  8.1198810278697D-07  1.4886381290124D-06  4.4119159005598D-06
  1.1183697066714D-06  2.6790899636017D-06  2.2398983449720D-06
  5.0426986042319D-07  2.5712236127425D-06  4.2133702926385D-06
  3.3938570905376D-06  4.8176122314913D-06  3.1369046314674D-06
  1.3368444106200D-06  4.7799158714321D-06  1.1289688481885D-06
  5.2249758859960D-06  4.8610721469495D-06  2.9432086331771D-06

OPTIMUN OBJECTIF FUNCTION
  1.6701964318793D-04
```

Table 7.50 Application 16: Output file of IPM on *cedar*

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

REORD	NCOEF	NCOEF2	Integer Memory	Real Memory	Total Memory
No Reord.	999,010	7,400,484	4,296,626	8,480,224	12,776,850
UnsyMMD	999,010	6,034,566	3,613,667	7,114,306	10,727,973

Table 7.51 HSCT FEM: Memory requirement for UNSYNUMFA

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

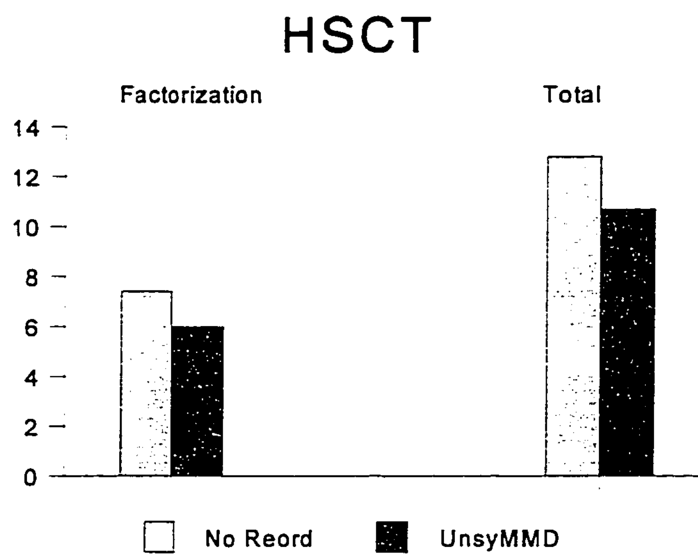


Fig. 7.31 HSCT: UNSYNUMFA. Non zero after factorization ($\times 10^6$)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Loop Unrolling Level	Symfa time (sec)	Numfa time (sec)	FBE time (sec)	Total time (sec)	Max. abs. displ.	Summat° abs. displ.	Relative Error Norm
1	0.480	50.010	0.310	53.350	0.447	301.291	1.34E-08
2	0.470	35.420	0.320	38.760	0.447	301.291	1.99E-08
8	0.480	28.730	0.320	32.700	0.447	301.291	1.36E-08

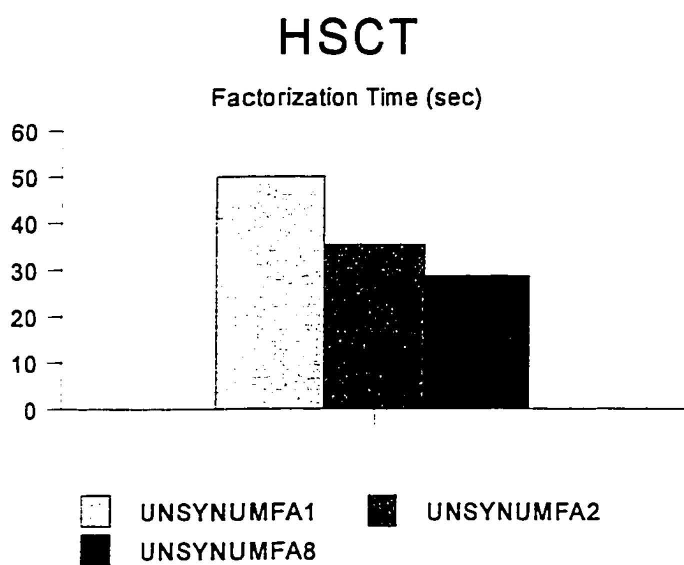
Table 7.52 HSCT FEM: Summary of results for UNSYNUMFA1/2/8 using UnsyMMD and different level of loop unrolling on the IBM RS6000/590 *Stretch* machine.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Loop Unrollin g Level	Symfa time (sec)	Numfa time (sec)	FBE time (sec)	Total time (sec)	Max abs. displ.	Summat° abs. displ.	Relative Error Norm
1	0.710	52.079	0.370	55.200	0.447	301.291	2.2E-09
2	0.680	35.650	0.380	38.730	0.447	301.291	2.0E-09
8	0.700	28.390	0.390	31.520	0.447	301.291	2.0E-09

Table 7.53 HSCT FEM: Comparison of results for UNSYNUMFA with no UnsyMMD and different level of loop unrolling on the IBM RS6000/590 *Stretch* machine.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



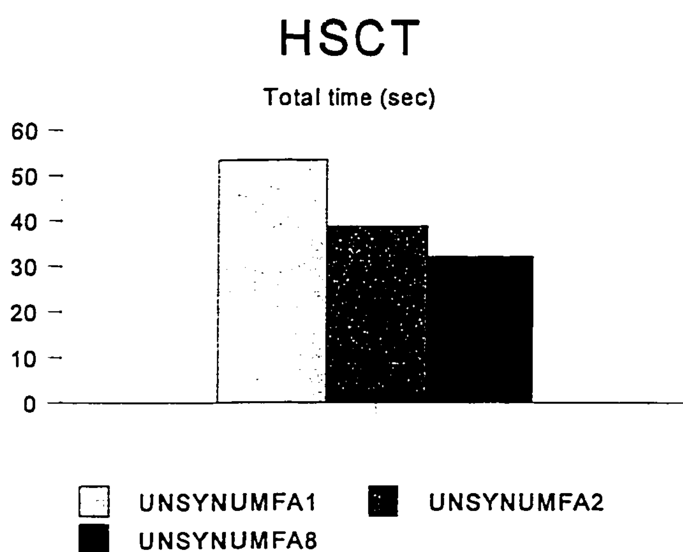
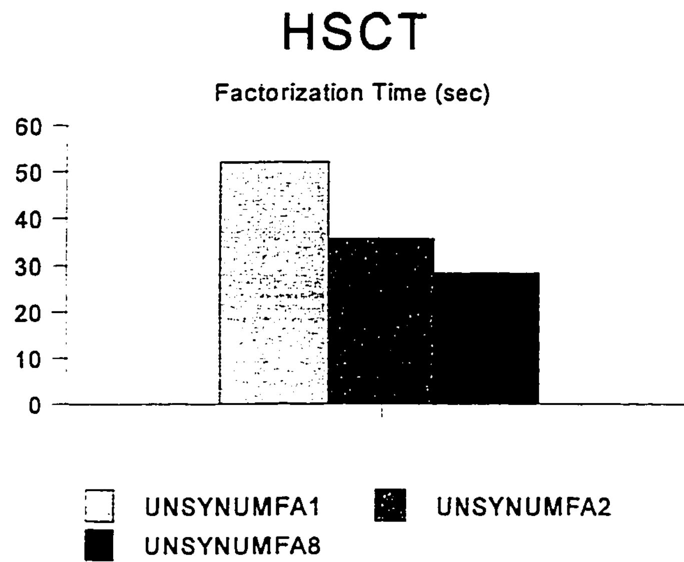


Fig. 7.32 HSCT FEM: Performance of UNSYNUMFA1/2/8 with UnsyMMD on the *stretch* machine

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



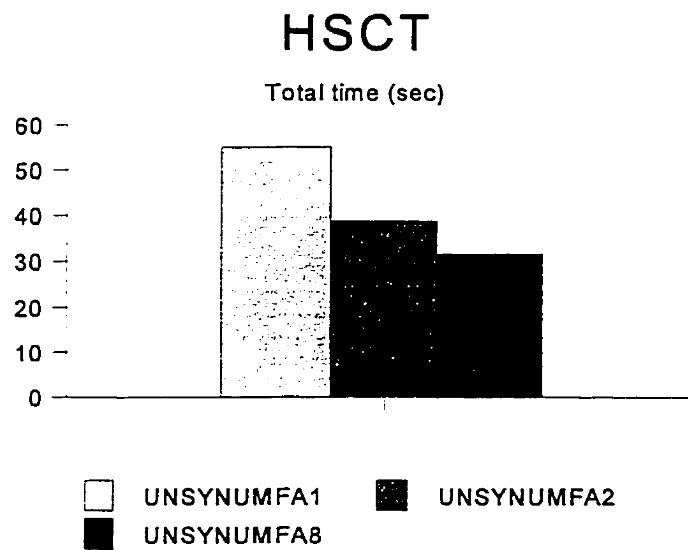


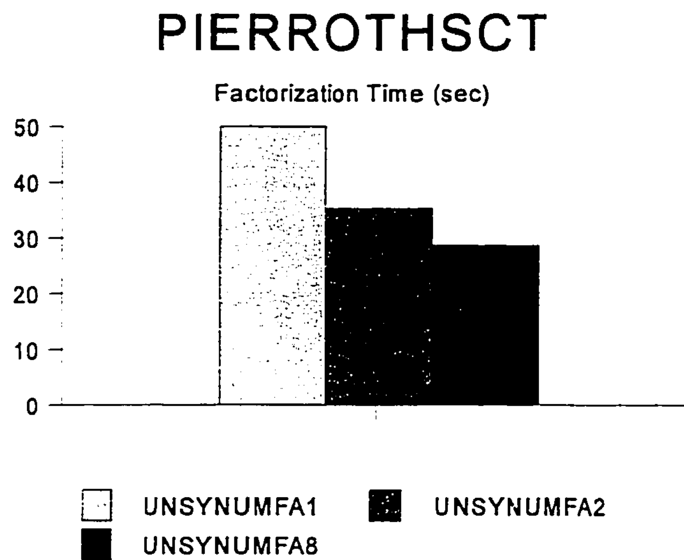
Fig. 7.33 HSCT FEM: Performance of UNSYNUMFA1/2/8 with no UnsyMMD on the *stretch* machine

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Loop Unrollin g Level	Symfa time (sec)	Numfa time (sec)	FBE time (sec)	Total time (sec)	Max abs. displ.	Summat° abs. displ.	Relative Error Norm
1	0.480	49.970	0.330	53.320	8.791	45.134	2.3E-07
2	0.470	35.340	0.320	38.650	8.791	45.134	1.8E-07
8	0.460	28.650	0.320	31.970	8.791	45.134	1.3E-07

Table 7.54 PierrotHSCT: Summary of results for UNSYNUMFA with UnsyMMD and different level of loop unrolling on the IBM RS6000/590 *Stretch* machine.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



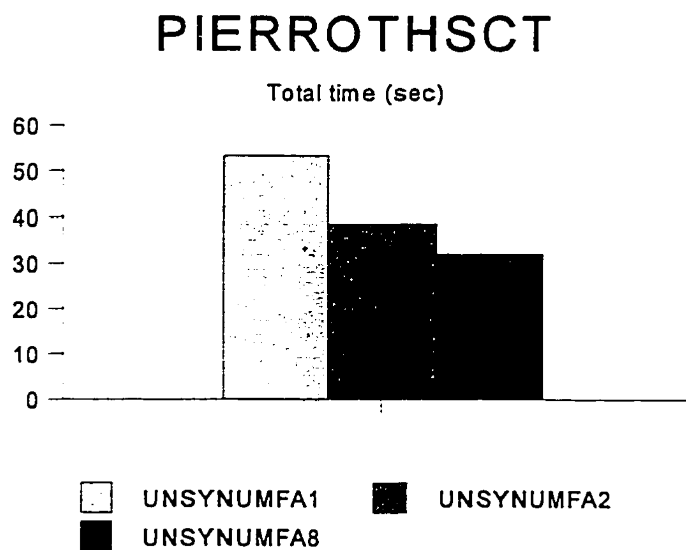


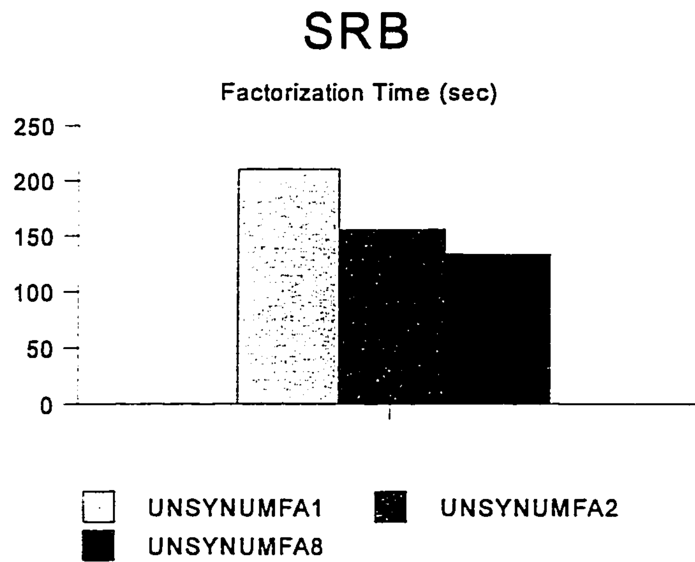
Fig. 7.34 PierrotHSCT: Summary of results of UNSYNUMFA1/2/8 with UnsyMMD on the *stretch* machine

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

LOOP Unrollin g Level	Symfa time (sec)	Numfa time (sec)	FBE time (sec)	Total time (sec)	Max abs. displ.	Summat° abs. displ.	Relative Error Norm
1	1.93	210.500	2.820	229.560	2.061	13569.65	8.1E-13
2	1.93	155.630	2.270	173.280	2.061	13569.65	8.1E-13
8	1.93	133.150	1.300	150.230	2.061	13569.65	8.1E-13

Table 7.55 SRB FEM: Summary of results for UNSYNUMFA using UnsyMMD and different level of loop unrolling on the IBM RS6000/590 *Stretch* machine.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



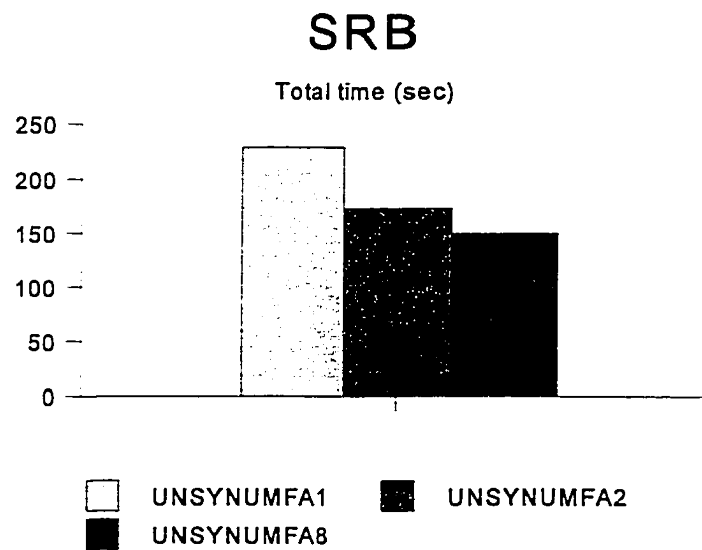


Fig. 7.35 SRB FEM: Performance of UNSYNUMFA1/2/8 with UnsyMMD on the *stretch* machine

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CHAPTER VIII

CONCLUSIONS AND SUGGESTIONS FOR FUTURE RESEARCH

8.1 Conclusions

Vector sparse solvers for positive, negative and indefinite systems have been developed. Efficient sparse technologies, such as: sparse symbolic factorization, sparse numerical factorization with unrolling strategies, sparse forward & backward solutions, sparse matrix-vector multiplication, have been developed, and fully utilized to improve the performance. The developed computer software has been fully optimized at the algorithm level, as well as during the compilation on vector computer platforms. The use of loop unrolling shows better results on high-performance uniprocessor computers. Efficient algorithms further reduce the amount of memory traffic on machines with high speed local memory, such as a cache. Large scale sparse matrices have been used to prove the robustness of the developed sparse equation solver for symmetric positive definite systems. Good performance has been achieved on the developed unsymmetrical solver for large scale applications.

Much of the research works in direct methods for the solution of sparse linear indefinite systems lies in determining the order in which pivots are chosen in the Gaussian elimination process, and how to minimize the fill-in during the factorization process. This choice can be made with a view to preserving sparsity, optimizing data structures, or maintaining stability. An alternative formulation and new computational strategies have been developed that satisfy all three requirements for solving general system of symmetrical and indefinite equations. Rotational matrix has been used to uncouple the 2x2 block

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

diagonal matrix, and therefore, greatly enhance the FORTRAN computer coding implementation. Mixed "backward factorization" and " Forward factorization" strategies have also been employed. The computational efficiency, and the solution accuracy have been validated by solving 5 indefinite system of equations (ranging from 51 to 15 367 unknown degree of freedom). Further numerical performance improvements have been realized by using MMD reordering algorithm (to minimize the number of fill-in) and by pushing all zero diagonal terms of the original matrix toward the bottom right of the matrix.

Major computational tasks in Subspace iterations, and Lanczos algorithms have been identified. Sparse Subspace and Lanczos eigensolvers for the solution of the generalized eigen-equations have been developed. Numerical results from practical finite element models have clearly indicated that the proposed sparse Subspace iterations, and Lanczos algorithms have offered substantial computational advantages over the traditional "skyline", or "variable bandwidth" strategies.

In this work, detailed discussions of a variation of the Karmarkar's Interior Point Method (IPM) have been presented. A Fortran implementation of the proposed method, using sparse technology, has been developed. Numerical examples to validate the entire procedure, and to show the promising potentials of using the IPM, in conjunction with efficient sparse indefinite solver, for solving linear programming problems have also been documented.

8.2 Suggestions for future research

Based upon the works that have been developed in this dissertation, the following future studies are suggested:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(1) Develop a parallel-vector sparse solver for positive definite systems. Appendix C gives some preliminary results of the implementation of a parallel sparse solver based on the substructuring formulation on Intel Paragon machines.

(2) Develop a vector sparse unsymmetrical solver, (unsymmetrical in locations and values) with pivoting and a reordering algorithm for a general unsymmetric matrix.

(3) Develop a callable sparse numerical library of subroutines for sequential and parallel-vector computers.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

REFERENCES

1. Bathe, J., Finite Element Procedures, Prentice-Hall, Englewood Cliffs, New Jersey (1996).

2. Tong, P., and Rossettos, J.N., Finite Element Method: Basic Technique and implementation, the MIT Press, Cambridge, Massachusetts, and London, England.

3. Hughes, T.J.R., The Finite Element Method, Prentice-Hall, Englewood Cliffs, N.J (1987)

4. Zinkiewicz, O.C., and Taylor, R.L., The Finite Element Method In Structural and Continuum Mechanics, McGraw-Hill, Vol. 1 and 2 (1989/1990).

5. Arora, J.S., Introduction to optimum design, McGraw-Hill, Inc. USA (1989).

6. Golub, G.H., and VanLoan, C.F., "Matrix Computations," 3rd Edition, The Johns Hopkins University Press (1996).

7. Bunch, J.R., and Kaufman, K., "Some Stable Methods For Calculating The Inertia and Solving Symmetric Linear Systems", Math. Comp., 31, pp. 162-179 (1977).

8. Duff, I.S., and Reid, J.K., "MA47: A Fortran Code For Direct Solution of Indefinite Sparse Symmetric Linear Systems", RAL Report #95-001 (Jan. 1995).

9. Nguyen, D.T., Storaasli, O.O., Carmana, E.A., Al-Nasra, M., Zhang, Y., Baddourah, M.A., and Agarwal, T.K., " Parallel-Vector Computation For Linear Structural Analysis and Nonlinear Unconstrained Optimization Problems", Computing Systems in Engineering, An International Journal, Vol. 2, No. 2/3, pp. 175-182 (Sept. 1991).

10. Duff, I.S., Erisman, A.M., and J.K.Reid, "Direct Methods For Sparse Matrices", Monographs On Numerical Analysis, Oxford Science Publications (1989).

11. George, J.A., and Liu, W.H., "Computer Solution of Large Sparse Positive Definite Systems", Preventive-Hall, Englewood Cliffs, N.J. (1981).

12. Belytscho, T., Plaskacz, E.J., Kennedy, J.M., and Greenwell, D.M, "Finite Element Analysis on the Connection Machine", Computer Methods Appl. Mech. Eng. 81, pp. 27-55 (1990).

13. John, Z., Hughes, T.J.R., Mathur, K.K., and Johnson, S.L., "A Data Parallel Finite Element Method For Computational Fluid Dynamics On The Connection Machine System", Comput. Methods Appl. Mech. Eng.. 99, pp. 113-134 (1992).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

14. Simon, H., Wu, P., and Yang, C., "Performance of a Supermodal General Sparse Solver on The Cray-YMP: 1.68 GFLOPS With Autotasking", Applied Mathematics Technical Report, Boeing Computer Services, SCA-TR-117 (March 1989).

15. Law, K.H., and Mackay, D.R., "A Parallel Row-Oriented Sparse Solution Method For Finite Element Structural Analysis", IJNM in Eng., Vol. 36, pp. 2895-2919 (1993).

16. Noor, A.K., "Parallel Processing In Finite Element Structural Analysis", in Parallel Computations and Their Impact On Mechanics, ASME, pp. 253-277, A.K. Noor (Ed.) (1987).

17. Khan, A.I., and Topping, B.H.V., "A Transputer Routing Algorithm For Nonlinear or Dynamic Finite Element Analysis", Engineering Computations, Vol. 11, pp. 549-564 (1994).

18. Zheng, D., and Chang, T.Y.P., "Parallel Cholesky Method On MIMD With Shared Memory", Computers & Structures, Vol. 56, No.1, pp. 25-38 (1995).

19. Chiang, K.N., and Fulton, R.E., "Structural Dynamic Methods For Concurrent Processing Computer," Computers & Structures, 36(6), pp. 1031-1037 (1990).

20. Qin, J., CE 795/895 CoursePack: "Sparse matrix technology", Civil and Environmental enginieering, Old Dominion University, Fall 1994.

21. Aminpour, M.A., Ransom, J.B., and McCleary, S.L., "A Coupled Analysis Method For Structures With Independently Modeled Finite Element Subdomains", IJNM in Eng.., Vol. 38, pp. 3695-3718 (1995).

22. Ransom, J.B., McCleary, S.L., and Aminpour, M.A.,"A New Interface Element For Connecting Independently Modeled Substructures", AIAA/ASME/ASCE/AHS SDM Conference Proceedings, AIAA-93-1503-cp (1993).

23. Housner, J.M., Aminpour, M.A., and McCleary, S.L., "Some Recent Developments In Computational Structural Mechanics", Proc. Int. Conf. Computational Engineering Science, ICES Publications, Atlanta, GA, pp. 376-381 (1991).

24. Ortega, J.M., Voigt, R.G., Solution of Partial Differential Equations on Vector and Parallel Computers, Society for Industrial and Applied Mathematics, 1985.

25. Ng., E.G., Peyton, B., "Block Sparse Cholesky algorithms on advanced uniprocessor computers", SIAM Journal Sci. Comput. Vol 14, No 5, pp. 1034-1056, September 1993.

26. Parlet, B.N., The Symmetric Eigenvalue Problem, Prentice-Hall, Englewood Cliffs, N.J. (1980).

27. Duff, I.S., Stewart, G.W., Sparse Matrix Proceedings, Siam , Philadelphia (1979).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

28. Gourlay, A.R., Watson, G.A., Computational Methods for Matrix Eigenproblems, John Wiley & Sons (1973).

29. Dongarra, J. and Hinds, A., Unrolling loops in FORTRAN, Software Pract. Exper., 9, pp. 219-229 (1979).

30. George, J.A., and Liu, W.H.,"Computer Solution of Large Sparse Positive Definite Systems" Prentice-hall, Englewood Cliffs, N.J. (1981).

31 Runesha, B.H., Nguyen, D.T., Belegundu, A.D. and Chandrupatla, T.R., "Interior Method with positive and indefinite sparse solvers for the linear programming optimal design problems", 4th NASA National Symposium on large-scale Analysis and Design on high-performance Computers and Workstations. Oct 15-17, 1997, williamsburg, VA.

32. Karmarkar, N., " A new polynomial-time Algorithm for linear programming", Combinatorica, 4, pp. 373-395 (1984).

33. Gill, P.E., Murray, W., Saunders, M.A., Tomlin, J.A., and Wright, M.H., "On projected Newton Barrier methods for linear programming and an Equivalence ot Karmarkar's projective Method", Mathematical programming, 36, 183-209 (1986).

34. Kranich, E., ' Interior Point Methods for Mathematical Programming: A bibliography", Discussion paper 171, Institute of economy and operations research, Fer Universitat Hagen, P.O.Box 940, D-5800 Hagen 1, Germany (1991).

35. Roos, C., and Vial, J. Ph., "Interior Point methods", in Advances in linear and Integer Programming, J.E. Beasley(ed.), Chapter 3, Oxford University Press, Oxford, England (1994).

36. Dantzig, G.B., " Linear Programming and Extensions", Princeton University press, Princeton, New-Jersey.

37. Chang, Saleeb, A.F., and Li, G., "Large Strain Analysis of Rubber-like Materials Based On a Perturbed Lagrangian Variational Principle," J. Comput. Mech., Vol. 8, pp.221-233 (1991).

38. Gunderson, "Fatigue Life of TLP Flex-elements," 24th Annual OTC Conference, Houston, Texas, May 4-7, 1992.

39. Gibbs, N. E., " A hybrid profile reduction algorithm", ACM Trans. On Math. Software, 2, pp. 378-387 (1976).

40. Bathe,K.J., "Solution Methods of Large Generalized Eigenvalue Problems in Structural Engineering, "Report UC SESM 71-20, Civil Engineering Department, University of California, Berkeley, 1971.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

41. Bathe, K.J., "Convergence of Subspace Iteration," in Formulations and Numerical Algorithms in Finite Element Analysis, MIT Press, Cambridge, MA, pp.575-598, 1977.

42. Bathe, and Ramaswamy, S, "An Accelerated Subspace Iteration Method", Computer Methods in Applied Mechanics and Engineering, Vol. 23, pp.313-331, 1980.

43. Bathe, and Wilson, E.L., "Eigensolution of Large Structural Systems with Small Bandwidth," ASCE Journal of Engineering Mechanics Division, Vol. 99, pp. 467- 479, 1973.

44. Lanczos, "An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operator," Journal of Research of the National Bureau of Standards, 45, pp. 255-281 (1950).

45. Golub, Underwood, R., and Wilkinson, J.H., "The Lanczos Algorithm for the Symmetric Ac = lBc Problem," Tech. Rept.. STAN-CS-72-720, Computer Science Department, Stanford University, 1972.

46. Parlett and Scott, D., "The Lanczos Algorithm with Selective Orthogonalization, "Mathematics of Computations, 33 No. 145, pp. 217-238 (1979).

47. Nour-Omid, Parlett, B.N., and Taylor, R.L., "Lanczos versus Subspace Iteration for solution of Eigenvalue Problems," International Journal for Numerical Methods in Engineering, 19, pp. 859-871 (1983).

48. Qin, J., Nguyen, D.T., and Zhang, Y., "A Parallel-Vector Lanczos eigensolver for Structural Vibration problems," in: Proceedings of the Fourth International Conference on Recent Advances in Structural Dynamics, July 15-18, 1991, London, UK.

49. Nguyen,D.T. Qin, J., Chang, T.P.Y. and Tong, P., "Efficient Sparse Equation Solver With Unrolling Strategies For Computational Mechanics," Proceedings of the ICES'97 conference, San Jose, Costa Rica (May 4-10, 1997).

50. Cuthill, E., and McKee, J., "Reducing The Bandwidth of Sparse Symmetric Matrices," Proceedings of 24th National Conference, Association for Computing Machinery, pp.157-172 (1969).

51. Gibbs, N.E., Poole, W.G., Stockmeyer, P.K. and Jr., "An Algorithm For Reducing the Bandwidth and Profile of a Sparse Matrix," SIAM Journal on Numerical Analysis, Vol. 13, pp. 236-250 (1976).

52. Lewis, Pfyton, B.W. and Pothen, A.,"A Fast Algorithm For Recording Sparse Matrices For Parallel Factorization," SIAM J. Sci. Statist. Comput., 6, pp. 1146-1173 (1989).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

53. Liu, "Reordering Sparse Matrices For Parallel Elimination," Tech. Report 87-01, Computer Science, York University, North York, Ontario, Canada (1987).

54. Storaasli, O.O., Nguyen, D.T., and Agarwal, T.K., "The Parallel Solution of Large-Scale Structural Analysis Problems on Supercomputers," AIAA Journal, Vol. 28, No.7, pp. 1211-1216 (July 1990).

55. Maker, B.N., Qin, J. and Nguyen, D.T., "Performance of NIKE3D with PVSOLVE On Vector and Parallel Computers," to appear in Computing Systems in Engineering Journal.

56. Wang, Chang, T.Y.P., and Tong, P., "Nonlinear Deformation Responses of Rubber Components by Finite Element Analysis," Computational Mechanics '95: Theory and Applications Proceedings of the International Conference on Computational Engineering Science. July 30-Aug. 3 '95, Hawaii, USA, Vol. 2, pp. 3135-3140.

57. Duff, I.S., Sparse Matrices and their Uses, Academic Press, New York (1981).

58. Zlatev, Z., Wasniewski, J., Schaumburg, K., Y12M Solution of Large and Sparse Systems of Linear Algebraic Equations, Springer-Verlag, Berlin (1981).

59. Bunch, J.R., Rose, D.J., Sparse Matrix Computations, Academic Press Inc. New York (1976).

60. Reid, J.K., Large Sparse Sets of Linear Equations, Academic Press, New York (1971).

61. Smith, B.T., Boyle, J.M., Garbow, Y. I., Kelma, V.C., Moler, C.B., Matrix Eigensystem Routines-EISPACK Guide, Springer-Verlag Berlin (1974).

62. Barnes, E.R., "A Variation of Karmarkar's Algorithm For Solving Linear Programming Problems", Mathematical programming, 36, pp. 174-182 (1986).

63. Chen, P., Runesha, H., Nguyen, D.T., Tong, P., Chang, T.Y.P., Chang, " Sparse Algorithms for Indefinite systems of linear Equations", proceedings of the ICES'97 conference, San Jose, Costa Rica (may 4-10, 1997).

64 Pissanetzky, S., "Sparse Matrix Technology", Academic Press Inc. (London) LTD (1984).

65 Golub, G.H., O'Leary, D.P., "Some history of the conjugate Gradient and Lanczos algorithms", SIAM Review, Vol 31, No 1, pp. 50-102 (1989).

66 Qin, J., Runesha, B.H. and Nguyen, D.T., "Enhancements of MA27 Code For Indefinite Sparse System of Equations", unpublished work (in preparation).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

67 Qin, J., Gray, Jr., C.E., Mei,C. and Nguyen, D.T., "A Parallel-Vector Equation Solver For Unsymmetric Matrices on Supercomputers", Computing Systems in Engineering, An International Journal, Vol.2, No. 2/3, September 1991 (Pergamon Press), pp.197-202.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX A

MULTI-PLATFORMS MAKEFILE

```
#*********************************************************************
#                 MAKEFILE                                           #
#                                                                    #
#           H. Runesha  June 30, 1997                                #
#                                                                    #
# This Makefile was inspired from the one written by                 #
# Michael Puso at Lawrence Livermore National Laboratory             #
#                                                                    #
# Look through makefile to comment and uncomment specific lines      #
# based on platform to compiled for SGI/DEC/SUN/HP/UNIX/CRAY         #
# CONVEX should be treated same as HP.                               #
# For example: you are compiling for a sun uncomment the solaris     #
# specific flags and make sure the other platform specific flags are #
# commented.                                                         #
#                                                                    #
# The name of the executable is: aa00                                #
#*********************************************************************
FORTRAN = ${FC}
FORTRAN = f77
CPP = /lib/cpp
CPP = /usr/ccs/lib/cpp
XLIB2 = -lX11
XLIB2 =
#-------------------------- SGI R8000 and up -------------------------#
# FFLAGS  = -O3 -static
# FFLAGS  = -g -static
# CPPFLAGS = -Dsgi
# SGI R4400 and down
# FFLAGS  = -O2 -static -mips2
# FFLAGS  = -O2 -static
# SGI debug
# FFLAGS  = -g -static

#--------------------------   Solaris   -------------------------#
 FFLAGS = -Bstatic -O3 -xcg92
 FFLAGS = -Bstatic -fast -xcg92 -O4 -Bstatic -xtarget=ultra
 FFLAGS = -Bstatic -O3 -xcg92
 CPPFLAGS = -Dsun
#--------------------------   DEC    --------------------------#
#FFLAGS = -O5 -static -cpp
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
#FFLAGS2 = -O4 -static -cpp
#CPPFLAGS = -Ddec
#--------------------------     CRAY     --------------------------#
# FORTRAN = ${CF}
# FFLAGS = -dp -ZP
# CPPFLAGS = -Dcray
#--------------------------     HP     --------------------------#
# FFLAGS =  +O3 -K +T +E1
# CPPFLAGS = -Dhpux
# LINK = +U77
#--------------------------     IBM     --------------------------#
# FFLAGS = -O3Q -qhssngl
# CPPFLAGS = -Dibm
# BIG_MEMORY = -bmaxdata:0x70000000
# XLIB2 = -lX11
#------------------------- ODU STRETCH - IBM -------------------------#
# FFLAGS = -LSP -O3 -qstrict -qalias=noaryovrlp -qarch=pwr2
# CPPFLAGS = -Dibm
# BIG_MEMORY = -bmaxdata:960000000
# XLIB2 =
#-------------------------------------------------------------------#
#Libraries ?
LIBS =
# Comment out line below for single precision version
DPFLAG = -DDP
#-------------------------------------------------------------------#
#
OBJS = \
	PierSpaSolver.o  main.o\
	reord.o
#-------------------------------------------------------------------#
aa00: ${OBJS}
	${FORTRAN} ${LINK} -o aa00  $(OBJS) $(LIBS)\
	 $(XLIB2) $(BIG_MEMORY)
#-------------------------------------------------------------------#
genb: ${LIBG}
	${FORTRAN} ${FFLAGS} ${LINK} -o genb genb_m.o $(LIBG)
# HP needs U77 library for timing routines (won't work if used on all routines)
# cputim.o: cputim.f
#	/lib/cpp -P ${CPPFLAGS} ${DPFLAG} cputim.f > cputim.F
#	${FORTRAN} ${FFLAGS} +U77 -c cputim.F
#	rm cputim.F
# DEC needs lower optimization compile flag for expand.f
#expand.o: #	${FORTRAN} -c ${FFLAGS2} ${DPFLAG} expand.f
.c.o:
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
	cc -c $?
# Comment and Uncomment the appropriate .f.o rules
#
# For SGI or DEC          use the following .f.o rules (2 lines)
#
# .f.o:
#      ${FORTRAN} ${FFLAGS} ${CPPFLAGS} ${DPFLAG} -c $?
#
# For SUN or HP or IBM    use the following .f.o rules (4 lines)
#
#      $(CPP) -P ${CPPFLAGS} ${DPFLAG} $<> $(*.F).F
.f.o:
	$(CPP) -P ${CPPFLAGS} ${DPFLAG}  $<> $*.F
	${FORTRAN} ${FFLAGS} -c  $*.F
	rm $*.F
# For CRAY                use the following .f.o rules (4 lines)
#.f.o:
#      cp $<  $(*.F).F
#      ${FORTRAN} ${FFLAGS} ${CPPFLAGS} ${DPFLAG} -c $(*.F).F
#      rm $(*.F).F
#${LIB}:      ${OBJS}
#      ar rv $@ $?
#${LIBG}: ${OBJSG}
#      ar rv genblib.a readk.o rmalloc.o rfree.o writeb.o
#      mv  readk.o ./GENB;  mv  rfree.o ./GENB;
#      mv  rmalloc.o ./GENB;  mv  writeb.o ./GENB
#      ar rv $@ $?
#      ranlib $@
clean:
	\rm -f ${OBJS} ${LIB}
	rm -f genb aa00 *.a
	rm ./GENB/*.o
#------------------------------------------------------------------------#
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX B

SUBROUTINE CPUTIME.F

```
      Subroutine cputime (time)
      Real tar(2)
      Real*8 time
c     -------- For IBM type machine --------------------------------
      time = 0.01*mclock()
c     -------- For SUN Workstations and other Unix boxes--------
c     time = etime (tar)
c     -------- For CRAY type machines ------------------------------
c     time = tsecnd()
      Return
      end
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX C

PARALLEL VECTOR SPARSE SOLVER

Given a matrix [K] and the right hand side vector {F} in NASA row wise format, let's consider the following system of linear equations

$$Kz = F \tag{C-1}$$

An algorithm for a parallel sparse solver has been suggested, based on the substructuring finite element formulation [1]. Each processor can either construct its assigned portion of the matrix associated to a substructure, or a given stiffness matrix [K] have to be rearranged into a V-shape form as shown in Fig.(C-1).

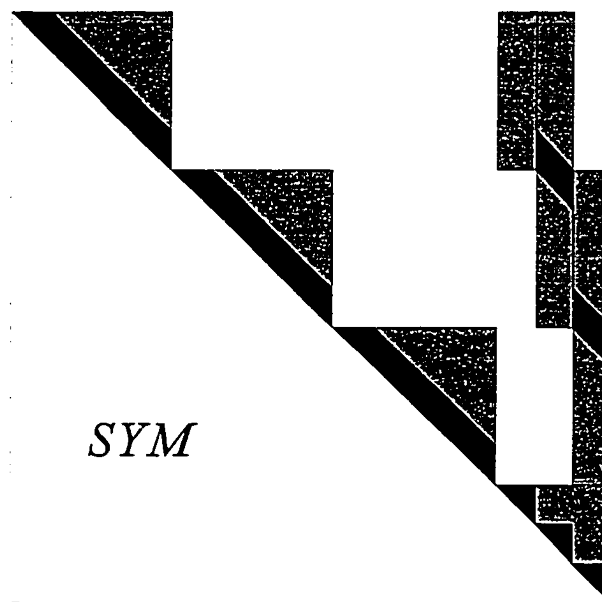


Fig. (C.1) Parallel sparse solver: V-shape form

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$\begin{bmatrix} K_{ii}^{(1)} & & & K_{ib}^{(1)} \\ & K_{ii}^{(2)} & & K_{ib}^{(2)} \\ & & K_{ii}^{(3)} & K_{ib}^{(3)} \\ K_{bi}^{(1)} & K_{bi}^{(2)} & K_{bi}^{(3)} & \sum K_{bb}^{(r)} \end{bmatrix} \begin{Bmatrix} Z_i^{(1)} \\ Z_i^{(2)} \\ Z_i^{(3)} \\ Z_b \end{Bmatrix} = \begin{Bmatrix} F_i^{(1)} \\ F_i^{(2)} \\ F_i^{(3)} \\ \sum F_b^{(r)} \end{Bmatrix}$$

Fig (C.2) Parallel sparse solver: Interior and Boundary displacements

The subscripts i and b correspond to the interior and boundary nodes, respectively. The submatrices $[K_{ib}^{(r)}]$ correspond to the coupling (boundary) submatrices. To solve for Equation (C-1) in parallel, two parallel sparse algorithms are required:

a) A parallel algorithm to rearrange the matrix into a V-shape that minimize the length of the coupling submatrices, and to perform the fill-in minimization. (When K is given).

b)The second algorithm is a parallel vector sparse solver for a V-shape matrix.

Let's consider the stiffness matrix in a V-shape form. The *boundary* displacement can be computed as follows:

$$\left[\sum K_{b_{eff}}^{(r)}\right]\{Z_b\} = \left\{\sum F_{b_{eff}}^{(r)}\right\} \qquad \textbf{(C-2)}$$

where

$$\left[\sum K_{b_{eff}}^{(r)}\right] = \sum K_{bb}^{(r)} - \sum K_{bi}^{(r)}[K_{ii}^{(r)}]^{-1} K_{ib}^{(r)} \qquad \textbf{(C-3)}$$

$$\left\{\sum F_{b_{eff}}^{(r)}\right\} = \sum F_b^{(r)} - \sum K_{bi}^{(r)}[K_{ii}^{(r)}]^{-1} F_i^{(r)} \qquad \textbf{(C-4)}$$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and the *interior* displacement can be computed from Eq.(C-5).

$$\{Z_i^{(r)}\} = [K_{ii}^{(r)}]^{-1}\{F_i^{(r)} - K_{ib}^{(r)}Z_b\} \quad \textbf{(C-5)}$$

The parallel vector sparse solver for V-shape matrices has been implemented on the Intel Paragon at NASA Langley, and completed on the Intel Paragon at the Hong Kong University of Science and Technology. The step by step algorithm of the parallel sparse solver is given in Table (C-1) and preliminary results that have been obtained are shown in Tables (C-2) and (C-3).

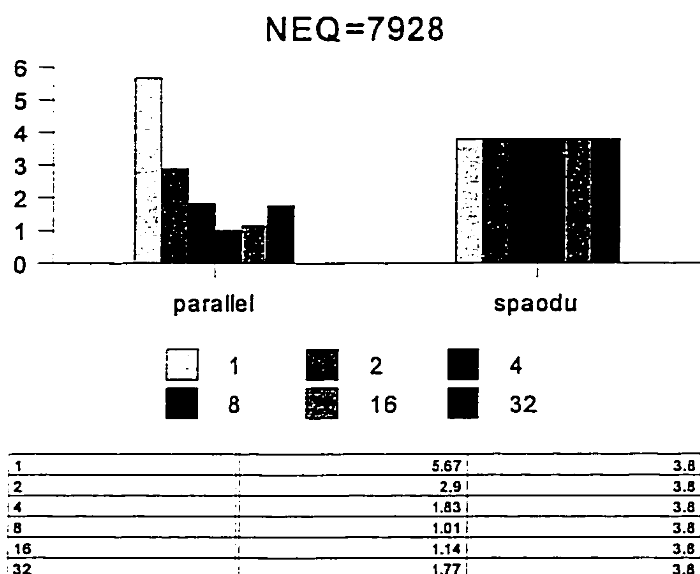
Step 1. Given $K_{ii}^{(r)}$, $F_i^{(r)}$ and $K_{ib}^{(r)}$ information of substructure r to processor r, in sparse format. r=1, number of processors
Step 2. Generate $\sum K_{bb}^{(r)}$ and $\sum F_b^{(r)}$
For each processor do:
Step 3. Symbolic factorization and find supernodes for $K_{ii}^{(r)}$ ==> 100% parallel
Step 4. (a) Numerical factorization. ==> 100% parallel
(b) Solve for $[K_{ii}^{(r)}]^{-1}*[K_{ib}^{(r)}]$ one column at the time
Call Forward/Backward and save result in a vector {x}.
(c) Perform $[K_{bi}^{(r)}]*\{x\}$
(d) Assemble $[\sum K_{b\,eff}^{(r)}]$
(e) Perform similar operations as in steps (b,c,d) to get $[\sum F_{b\,eff}^{(r)}]$
Step 5. Solve for boundary displacement Eq.(C-2)
Step 6. Solve for interior displacements. Eq. (C-5)

Table (C-1) Parallel sparse solver: Step by step algorithm

The symbolic and numerical factorization in steps 3 and 4 uses the vector sparse solver developed in Chapter II.

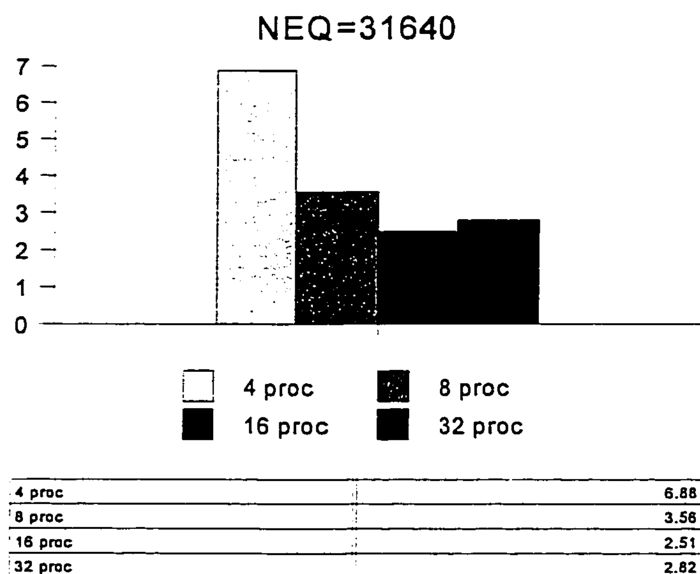
Results in Tables (C-2) and (C-3) shows that parallel speed-up can be achieved for a matrix already in V-shape(from substructure formulation) with a small coupling bandwidth.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



	parallel	spaodu
1	5.67	3.8
2	2.9	3.8
4	1.83	3.8
8	1.01	3.8
16	1.14	3.8
32	1.77	3.8

Table (C-2) PVS-solver: summary of results of NEQ= 7928



4 proc	6.88
8 proc	3.56
16 proc	2.51
32 proc	2.82

Table (C-3) PVS-solver: summary of results of NEQ= 31640

Future tasks include, the migration of the developed code to a new parallel platform (such as IBM SP2) and the development of a parallel matrix partition algorithm (for the V-shape form) that minimize the fill-in.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

VITA

Hakizumwami Birali Runesha was born in Bujumbura, Burundi on June 27, 1965. In October 1989, he graduated with honor from the college of engineering at the University of Kinshasa, Zaire, with a Bachelor of Engineering Science. He worked as a faculty member in the Civil Engineering Department of the University of Kinshasa before winning a scholarship for graduate school in the United States in 1990. He joined Old Dominion University (ODU) in 1991 and earned is Master of Science Degree in 1993. The same year, he joined the Ph.D. program at ODU and became a Ph.D. candidate in 1995. During his study at ODU, he worked on projects granted by the National Aeronautics and Space Administration (NASA) at Langley Research Center, lectured engineering courses and was appointed as a Visiting scholar at the Hong Kong University of Science and Technology (HKUST). He is a member of ASCE and IEEE. He is a member of Phi Kappa Phi and Chi Epsilon honor societies. He was in the Who's Who among students in American universities and colleges in 1996. He has published the following articles:

"Interior method with positive and indefinite sparse solvers for linear programming optimal design problems," by B.H. Runesha, D.T. Nguyen, A.D. Belegundu and T.R. Chandrupatla. Proceedings of the 4th NASA national symposium on large-scale analysis and design on high-performance computers and workstations. Oct 15-17, 1997, Williamsburg, VA.

"Automatic differentiation for design sensitivity analysis of structural systems using Parallel-vector processor," by D.T. Nguyen, R. Qamar and B.H. Runesha. Proceedings of the 4th NASA national symposium on large-scale analysis and design on high-performance computers and workstations. Oct 15-17, 1997, Williamsburg, VA.

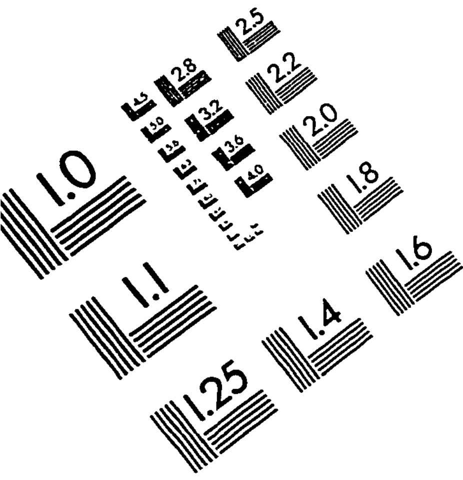
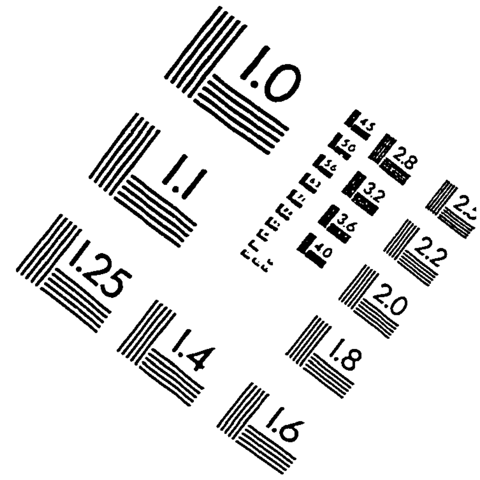
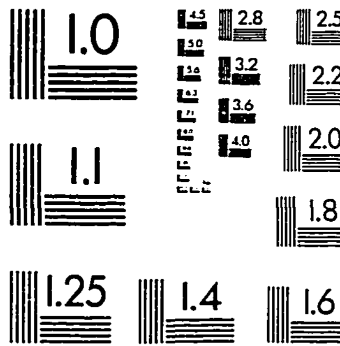
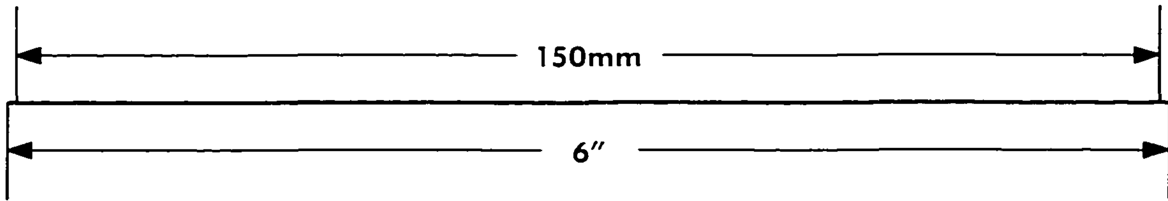
"Subspace iteration and vector-sparse technology for generalized eigen-value problems," by B.H. Runesha, D.T. Nguyen, P.Tong, T.Y.P Chang. Proceedings of the International Conference on Computational Engineering Science, ICES 97. May 4-9, 1997, San Jose, Costa Rica.

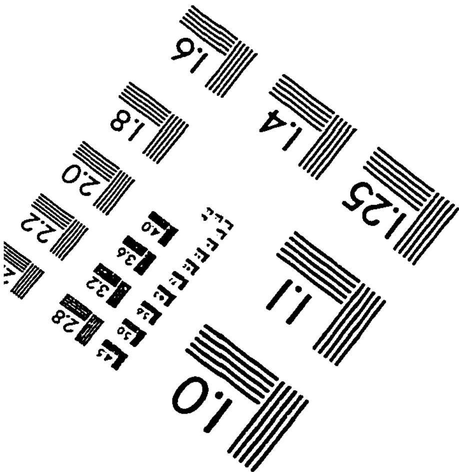
"Sparse algorithms for indefinite system of linear equations," by P. Chen, B.H. Runesha, D.T. Nguyen, P.Tong, T.Y.P Chang. Proceedings of the International Conference on Computational Engineering Science, ICES 97. May 4-9, 1997, San Jose, Costa Rica.

"Parallel finite element matrix assembly and equation solver using PVM on cluster of workstations," by B.H. Runesha. Submitted to the Robert J. Melosh medal competition at the school of engineering, Duke University, Durham, North Carolina, December 30, 1995.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

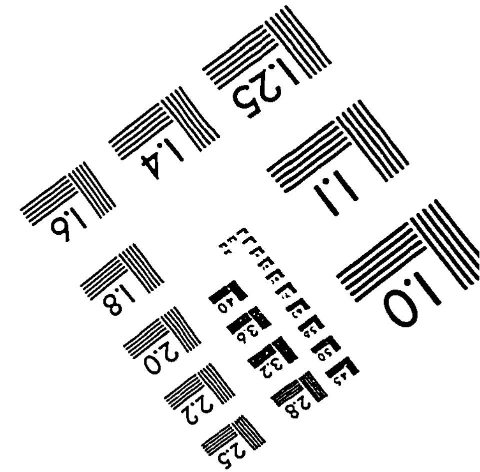
IMAGE EVALUATION TEST TARGET (QA-3)



APPLIED IMAGE . Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989

© 1993, Applied Image, Inc., All Rights Reserved



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.