Summer 2012

Efficient Stand-Alone Generalized Inverse Algorithms and Software for Engineering/Sciences Applications: Research and Education

Subhash Chandra Bose S V Kadiam
Old Dominion University

EFFICIENT STAND-ALONE GENERALIZED INVERSE ALGORITHMS AND SOFTWARE FOR ENGINEERING/SCIENCES APPLICATIONS: RESEARCH AND EDUCATION

by

Subhash Chandra Bose S V Kadiam
B.E. May 2004, Osmania University, India
M.Tech. June 2006, Acharya Nagarjuna University, India

A Dissertation Submitted to the Faculty of Old Dominion University
in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

CIVIL AND ENVIRONMENTAL ENGINEERING

OLD DOMINION UNIVERSITY
August 2012

Approved by:

Duc T. Nguyen (Director)

ManWo Ng (Member)

Yaohang Li (Member)

Julie Zhili Hao (Member)

ABSTRACT

EFFICIENT STAND-ALONE GENERALIZED INVERSE ALGORITHMS AND SOFTWARE FOR ENGINEERING/SCIENCES APPLICATIONS: RESEARCH AND EDUCATION

Subhash Chandra Bose S V Kadiam
Old Dominion University, 2012
Director: Dr. Duc T Nguyen

Efficient numerical procedures for finding the generalized (or pseudo) inverse of a general (square/rectangle, symmetrical/unsymmetrical, non-singular/singular, real/complex numbers) matrix and solving systems of Simultaneous Linear Equations (SLE) are formulated and explained. The developed procedures and its associated computer software (under MATLAB computer environment) have been based on "special Cholesky factorization schemes" (for a singular matrix), the generalized inverse of the matrix product, and were further enhanced by the Domain Decomposition (DD) formulation.

Test matrices from different fields of applications have been chosen, tested and compared with other existing algorithms. The results of the numerical tests have indicated that the developed procedures are far more efficient than existing algorithms.

Furthermore, an educational version of the generalized inverse algorithms and software for solving SLE has also been developed to run any FORTRAN and/or 'C' programs over the web. This developed technology and software is freely available and can run on any device with internet connectivity and browser capability.

This dissertation is dedicated to my father.

ACKNOWLEDGMENTS

I am extremely grateful to the people who, directly or indirectly, helped to make this research possible. Firstly, I would like to express my deepest gratitude to my advisor, Professor Duc T. Nguyen, for his patient guidance, encouragement and valuable discussions during the course of this research. I have been very fortunate to have an advisor who helped me overcome many difficult situations and finish this dissertation. It would not have been possible to accomplish this work without his constant support.

I would like to thank the other members of my committee Dr. Julie Zhili Hao, Dr. Yaohang Li and Dr. ManWo Ng for their valuable comments and suggestions. I would also like to thank Mr. Amit H. Kumar from OCCS, ODU for his support.

A special thanks to all the graduate students from Civil and Environmental Engineering department for creating excellent working conditions.

Most importantly, I would like to thank my family, who have been supportive throughout my life and academic career.

NOMENCLATURE

SVD	Singular Value Decomposition
SLE	Simultaneous Linear Equations
SPD	Symmetric Positive Definite
GMRES	Generalized Minimal Residual
CG	Conjugate Gradient Algorithm
DD	Domain Decomposition Formulations
LDL^T	LDL Transpose Algorithm
nsu	Number of Sub-domains
$pinv()$	MATLAB Command to Compute Generalized Inverse
$inv()$	MATLAB Command to Compute Regular Inverse
A^+	Generalized Inverse of A
CG	Conjugate Gradient Algorithm
$Bi-CG$	Bi-Conjugate Gradient Algorithm
$GMRES$	Generalized Minimal Residual Algorithm
PCG	Preconditioned Conjugate Gradient Algorithm
$ODU-ginverse$	ODU Generalized Inverse Solver
$geninv$	Generalized Inverse Algorithm Discussed in [13]
$MATLAB-pinv$	MATLAB Generalized Inverse Solver
$ODU-ginverse\ iterative$	CG Iterative Method in ODU Generalized Inverse Solver

$ODU - DD$	ODU Generalized Inverse Domain Decomposition Solver
$Original\ System\ ginverse\ MV$	ODU Generalized Inverse Solver
$Original\ System\ geninv$	Generalized Inverse Algorithm discussed in [13]

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

Page

Page

Page

LIST OF FIGURES

Figure Page

Page

1. INTRODUCTION

In scientific computing, most computational time is spent on solving systems of Simultaneous Linear Equations (SLE) which can be represented in matrix notations as

$$Ax = b \tag{1.1}$$

where $A \in R^{n \times n}$ is a singular/non-singular matrix, and b is a given vector in R^n. For practical engineering/science applications, matrix A can be either sparse (for most cases), or dense (for some cases). Solving large scale system of SLE has been (and continues to be) a major challenging problem for many real-world engineering and science applications.

The generalized (or pseudo) inverse of a matrix is an extension of the ordinary/regular square (non-singular) matrix inverse, which can be applied to any matrix (such as singular, rectangular, etc.). The generalized inverse has numerous important engineering and science applications. Over the past decades, generalized inverses of matrices and their applications have been investigated by many researchers [1-8].

1.1 Literature Survey

Various methods have been proposed for finding the generalized inverse and its associated SLE. Xuzhou Chen et. al. [9] has proposed a method based on a finite recursive algorithm. The approach was based on the symmetric rank-one update. The algorithm proposed by Xuzhou Chen, however, was inefficient (in terms of computational time) and requires lot of computer memory. It has been shown that this algorithm [9] can be only effective for the computation of generalized inverse/Moore-

Penrose inverse of rectangular matrices (with rows<<cols or cols<<rows) and is inefficient for square matrices.

The most commonly implemented method in programming languages to compute generalized inverse (and its associated SLE) was based on Singular Value Decomposition (SVD) [3, 10-11]. This method is numerically very stable, however, it is computationally expensive for practical applications. MATLAB [12] uses SVD to compute the pseudo/generalized inverse by invoking the built-in function *pinv()*. It should be noted here that in finding the solution for SLE (with square/singular, or rectangular coefficient matrices), MATLAB and most (if not all) other researchers have computed the generalized inverse explicitly. Then, the solution can be found by a simple matrix times vector operation.

Since the standard Eigen-value problems (of $A \times A^H, and\ A^H \times A$) need to be solved in SVD, this method is computationally expensive. Despite of this fact, solving Eq. (1.1) by using SVD is still more efficient than Xuzhou Chen's proposed finite recursive algorithm [9].

In [6], an efficient algorithm for finding the generalized inverse of a (full rank) rectangular (or square) matrix has been proposed. However, this algorithm has not been able to handle the cases where the matrix has rank deficiency (such as a matrix which has some dependent rows and/or columns)

Pierre Courrieu [13] has proposed an algorithm to explicitly compute the generalized inverse using full-rank Cholesky factorization on the coefficient matrix. His algorithm was based on a theorem to compute the generalized inverse of a product of two matrices. Pierre Courrieu's algorithm has proven to be more efficient than finite recursive method [9] and SVD [10-11].

1.2 SVD and the Generalized Inverse

A general (square or rectangular) matrix $A \in R^{n \times n}$ can be decomposed as

$$A = U\Sigma V^H \tag{1.2}$$

where

$$\Sigma = a\ diagonal\ matrix\ (does\ NOT\ have\ to\ be\ a\ square\ matrix)$$

$$= \begin{cases} \Sigma_{ij} = 0, for\ i \neq j \\ \Sigma_{ij} \geq 0, for\ i = j \end{cases} \tag{1.3}$$

$$[U] and\ [V] = unitary\ matrices$$

$$and \begin{Bmatrix} U^H = U^T (for\ real\ matrices) \\ U^H = U^{-1} \end{Bmatrix} \tag{1.4}$$

Let A be a singular matrix of size $m \times n$ and let k be the rank of the matrix. Based on Eq. (1.2), one has

$$A = U\Sigma V^H;$$

$$where\ \Sigma = \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_k \end{bmatrix}$$

with $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_k > 0;$ (1.5)

and $\sigma_i = \sqrt{Eigen - Values\ of\ A^T A\ (or\ AA^T)}$

<u>Note:</u> Eigen-values of $A \times A^H$ and Eigen-values of $A^H \times A$ are the same. However, the Eigen-vectors of $A \times A^H$ and Eigen-vectors of $A^H \times A$ are "NOT" the same.

Then, the generalized inverse A^+ of A is the $n \times m$ matrix and is given as

$$A^+ = V\Sigma^+ \mathrm{U}^\mathrm{H} \tag{1.6}$$

where

$\Sigma^+ = \begin{bmatrix} [E] & [0] \\ [0] & [0] \end{bmatrix}$ and E is the $k \times k$ diagonal matrix, with

$E_{ii} = \Sigma_i^{-1}\ for\ 1 \leq i \leq k$

More details about computing the SVD from a given matrix $[A]$ can be found in the Appendix A.

1.3 Objective

The main objective of this dissertation is to develop an efficient (in terms of computational time and computer memory requirement) generalized inverse formulation to solve SLE with full or deficient rank of the coefficient matrix. The coefficient matrix can be singular/non-singular, symmetric/unsymmetric, square/rectangular, and with real/complex numbers. The proposed generalized inverse procedures can also be integrated in to the Domain Decomposition (DD) formulation for solving general, large scale SLE commonly encountered in engineering/sciences applications. Due to popular MATLAB software, which is widely accepted by researchers and educators worldwide, the developed code from this work is written in MATLAB language and has the following capabilities/features:

a) A stand-alone generalized inverse software to solve SLE

b) A stand-alone DD generalized inverse software to solve SLE

c) Utilizing sparse storage scheme (whenever possible) for storing data and solving SLE

d) Developing user friendly interfaces to test new problems (including the numerical data downloaded from popular web sites [14-15]).

e) Additional wall-time reduction for DD generalized inverse solver can be realized/achieved by performing "parallel matrix times matrix" operations under MATLAB-MPI computer environment.

f) Developing an "educational version" (written in FORTRAN language) of the software, which uses the generalized inverse for solving general SLE on the internet.

In Chapter 2, some major algorithms for solving SLE by direct and iterative methods are reviewed. These methods are mainly designed for solving non-singular SLE. Simple/basic domain decomposition (DD) algorithms, using mixed direct-iterative solvers, are discussed in Chapter 3. Major works in this dissertation are presented in Chapter 4, where efficient "generalized (or pseudo-) inverse" algorithms are thoroughly explained with and without incorporating the DD formulation. The numerical performance of the proposed algorithms are conducted in Chapter 5, through extensive set of coefficient matrices (including rectangular, square, symmetrical, non-symmetrical, singular, non-singular matrices) obtained from well established/popular websites [14-15]. Detailed procedures for executing any FORTRAN code (such as the "educational version" of the developed generalized inverse code for solving SLE) on the internet are explained and demonstrated in Chapter 6. Basic/simple parallel MATLAB-MPI functions (including parallel matrix times matrix operations) are summarized in Chapter 7. Finally, conclusions and future research works are summarized in Chapter 8.

2. DIRECT AND ITERATIVE METHODS FOR SYSTEM OF NON-SINGULAR SLE

Many real life, practical problems in scientific computing require efficient solution of Simultaneous Linear Equations (SLE), which can be conveniently expressed in the matrix notation as

$$Ax = b \tag{2.1}$$

In general, solutions for Eq. (2.1) can be classified into 2 categories: Direct and Iterative methods. In the subsequent sections, basic ideas behind these two types of solution approaches will be briefly summarized and discussed.

2.1 Direct Methods for Solving SLE

Depending on the nature of the coefficient matrix A, shown in Eq. (2.1), different direct methods/algorithms are available, such as:

a) Cholesky algorithm [if matrix A is Symmetric Positive Definite (SPD)]

b) LDL^T algorithm [if matrix A is Symmetric, could be either positive or negative definite]

c) LU decomposition algorithm [if matrix A is unsymmetric]

2.1.1 Cholesky Method

If the coefficient matrix A is Symmetric Positive Definite (SPD), then the following three step Cholesky algorithm can be used to obtain the solution for Eq. (2.1)

Step1: Matrix Factorization Phase

The coefficient matrix A is decomposed into

$$[A] = [U]^T U \tag{2.2}$$

where U is an $n \times n$ upper triangular matrix. For a general $n \times n$ SPD $[A]$, the diagonal and off-diagonal terms of the factorized matrix U can be computed from the following formulas [3, 10-11]

$$u_{ii} = \left(A_{ii} - \sum_{k=1}^{i-1} (u_{ki})^2 \right)^{\frac{1}{2}}$$

and

$$u_{ij} = \frac{A_{ij} - \sum_{k=1}^{i-1} u_{ki} u_{kj}}{u_{ii}}$$

Note: Since the "square root" operation is required for computing the diagonal terms of U, positive definite is a requirement for matrix A to assure the number under the square root is positive.

Step2: Forward solution phase

Substituting Eq. (2.2) in Eq. (2.1)

$$[U]^T [U]\{x\} = \{b\} \tag{2.3}$$

Let's define

$$[U]\{x\} = \{y\} \tag{2.4}$$

Eq. (2.3) becomes

$$[U]^T \{y\} = \{b\} \tag{2.5}$$

The intermediate unknown $\{y\}$ can be easily solved from Eq. (2.5) and hence the name "forward solution".

Step3: Backward solution phase

From Eq. (2.4), the unknown vector $\{x\}$ can be effectively solved and hence the name "backward solution".

The matrix factorization phase (step 1) is the most-time consuming part of solving SLE in the Cholesky algorithm. However, if the right-hand-side (RHS) vector $\{b\}$, shown in Eq. (2.1), becomes a matrix (with multiple columns), then the combined Forward and Backward solution time may become significant (as compared to the matrix factorization phase). As a general rule of thumb, computational time/effort for one matrix factorization is roughly equivalent to 20-25 times the efforts for one Forward and Backward solution phases.

2.1.2 LDL^T Method

In many engineering and science applications, the coefficient matrix in Eq. (2.1) is symmetric, however it may not be positive definite. The coefficient matrix could be negative definite. In this case, LDL^T algorithms can be used for solving Eq. (2.1), which also requires the following 3 computational steps

Step1: Matrix Factorization phase

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

In Eq. (2.6), matrices $[L]$ *and* $[D]$ represent the lower triangular (with 1 on its diagonal) and diagonal matrices, respectively.

Step2: Forward Solution and Diagonal Scaling phase

Substituting Eq. (2.6) in Eq. (2.1), one gets

$$[L][D][L]^T\{x\} = \{b\} \tag{2.7}$$

Let's define

$$[L]^T\{x\} = \{y\} \tag{2.8}$$

$$[D]\{y\} = \{z\} \Leftrightarrow \begin{bmatrix} D_{11} & 0 & 0 \\ 0 & D_{22} & 0 \\ 0 & 0 & D_{33} \end{bmatrix} \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \end{Bmatrix} = \begin{Bmatrix} z_1 \\ z_2 \\ z_3 \end{Bmatrix} \tag{2.9}$$

or, $y_i = \frac{z_i}{D_{ii}}; for\ i = 1,2,3, \dots N$ (2.10)

Then Eq. (2.7) becomes:

$$[L]\{z\} = \{b\} \Leftrightarrow \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \begin{Bmatrix} z_1 \\ z_2 \\ z_3 \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ b_3 \end{Bmatrix} \tag{2.11}$$

or, $z_i = b_i - \sum_{k=1}^{i-1} L_{ik} z_k; for\ i = 1,2,3, \dots . N$ (2.12)

Step3: Backward Solution phase

In this step, Eq. (2.8) can be effectively solved for the original unknown vector $\{x\}$

$$[L]^T\{x\} = \{y\} \Leftrightarrow \begin{bmatrix} 1 & L_{21} & L_{31} \\ 0 & 1 & L_{32} \\ 0 & 0 & 1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \end{Bmatrix} = \begin{Bmatrix} y_1 \\ y_2 \\ y_3 \end{Bmatrix} \tag{2.13}$$

or, $x_i = y_i - \sum_{k=i+1}^{N}; for\ i = N, N-1, \dots 1$

2.1.3 LU Decomposition Method

LU decomposition can be used to solve Eq. (2.1), when the coefficient matrix A is unsymmetric.

Step1: Factorization phase

The coefficient matrix in Eq. (2.1) can be factorized as a product of two matrices

$$A = L.U \tag{2.14}$$

where L is lower triangular (with values 1 on its diagonal) and U is upper triangular.

For the case of 4×4 matrix A. Eq. (2.14) would look like

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \alpha_{21} & 1 & 0 & 0 \\ \alpha_{31} & \alpha_{32} & 1 & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & 1 \end{bmatrix} \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix} \quad (2.15)$$

Various terms inside matrices $[L]$ *and* $[U]$ can be computed by equating both sides of Eq. (2.15).

Step2: Forward solution phase

Substituting Eq. (2.14) into Eq. (2.1), one obtains

$$[L.U]x = b \quad (2.16)$$

Let us define

$$[U]\{x\} = y \quad (2.17)$$

Substituting Eq. (2.17) in Eq. (2.16), one gets

$$[L]\{y\} = b \quad (2.18)$$

In this "forward solution" phase, Eq. (2.18) can be easily solved for the "intermediate" unknown vector $\{y\}$.

Step3: Backward solution phase

Solving the unknown vector $\{x\}$ in Eq. (2.17) is called "backward solution" phase

$$[U]\{x\} = y \quad \text{(2.17, repeated)}$$

While direct methods have offered advantages in terms of its robustness, accuracy, and reliability, etc., for large/sparse SLE (especially for 3-D problems), these direct methods may become excessively expensive. Furthermore, direct methods also have the following limitations:

a) The amount of required computer memory can be high.

b) The operation counts can be high, especially when many non-zero fill-in terms occurred during the factorization phase, even though reordering algorithms have commonly used (to minimize the non-zero fill-in terms) prior to numerical factorization phase, and

c) These methods have low degree of parallelism (or not easy to parallelize).

The above-mentioned drawbacks have motivated researchers to investigate iterative methods as possible alternative choices.

2.2 Iterative Methods for Solving SLE

Iterative methods can be superior to direct methods in all the above mentioned three aspects. Some of the popular iterative methods are Conjugate Gradient (CG), Bi-Conjugate Gradient (Bi-CG) with or without stabilizers [1, 3, 10], Generalized Minimal Residual (GMRES) [1, 3, 10], etc. These methods (CG for symmetrical, while Bi-CG and GMRES for unsymmetrical systems of SLE) can lead to low memory requirement and make effectively use of parallelism. Most (if not all) existing iterative algorithms require "matrix times vector" and "dot product of 2 vectors" operations. For these reasons, iterative methods are much more easier to parallelize (for improving computational efficiency) as compared to direct methods. These advantages make iterative linear system solvers as attractive alternatives to direct methods, particularly for large (3-D) problems. Despite of these desirable features, iterative methods may also have difficulties for fast convergence (or even have divergence) to a specified (small) error tolerance etc…, unless these iterative methods were used in conjunction with "efficient preconditioned" algorithms [1-3, 10-11] !

2.2.1 Conjugate Gradient (CG) Algorithm with Preconditioner

For systems of Symmetrical Positive Definite (SPD) SLE, the Preconditioned Conjugate Gradient (PCG) algorithms can be considered as the method of choice. PCG algorithms can be summarized in the following step-by-step numerical procedures [3, 10-11], for solving Eq. (2.1)

Eq. (2.1) can be re-casted as:

$$PAP^T P^{-T}\vec{x} = P\vec{b} \tag{2.19}$$

Eq. (2.19) can be expressed in the following general form [1, 3, 10-11]:

$$[A^*]\vec{y} = \overrightarrow{b^*} \tag{2.20}$$

where matrix

$[A^*] = [P] \times [A] \times [P^T]$ = symmetrical matrix, and the right-hand-side vector $\{b^*\}$ is defined as

$$\{b^*\} = [P] \times \{b\} \tag{2.21}$$

$$\vec{y} = P^{-T}\vec{x} \tag{2.22}$$

and $[P]$= preconditioned matrix

The step-by-step PCG algorithm is summarized in Fig. 2.1

Given the initial guessed vector $\vec{x}^{(0)}$

Compute (or input) the preconditioned matrix $[P]$

Compute $\vec{r}^{(0)} = \{Pb\} - [PAP^T]\vec{x}^{(0)}$

Set $\vec{d}^{(0)} = 0; \rho_{-1} = 1$

Do $i = 1,2,..$

$$\rho_{i-1} = \{r^{(i-1)}\}^T \{r^{(i-1)}\}$$

$$\beta_{i-1} = \frac{\rho_{i-1}}{\rho_{i-2}}$$

$$d^{(i)} = r^{(i-1)} + \beta_{i-1} d^{(i-1)}$$

$$q^{(i)} = [PAP^T] d^{(i)}$$

$$\alpha_i = \frac{\rho_{i-1}}{\{d^{(i)}\}^T \{q^{(i)}\}}$$

$$x^{(i)} = x^{(i-1)} + \alpha_i d^{(i)}$$

$$r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$$

Converge??

End do

If converged, then set

$\vec{x} = [P]^T \vec{x}$

Figure 2.1 Preconditioned Conjugate Gradient Algorithm

2.2.2 GMRES Algorithm [1, 3, 10, 16]

For systems of "unsymmetrical" SLE, popular algorithms such as GMRES, Bi-Conjugate Gradient (Bi-CG), Bi-Conjugate Gradient with Stabilizers (Bi-CG Stab) [1, 3, 10, 16] are recommended. For readers' convenience, a version of GMRES algorithms can be summarized in the following step-by-step numerical procedures [1, 3, 11], for solving $[A^*]\vec{y} = \overrightarrow{b^*}$

$$
\begin{aligned}
& r_0 = b - Ax_0 \\
& v_1 = \frac{r_0}{\|r_0\|_2} \\
& start \\
& for\ j = 1:m \\
& \quad w = Av_j \\
& \quad for\ i = 1:j \\
& \quad\quad H(i,j) = (w, v_i) \\
& \quad\quad w = w - H(i,j)v_i \\
& \quad end \\
& \quad v_{j+1} = \frac{w}{\|w\|_2} \\
& \quad if\ \|v_{j+1}\|_2 < tolerance \quad Break \\
& end \\
& V_{j+1} = [v_1, v_2, v_3, \ldots, v_{j+1}] \\
& H = V_j^T A V_j \\
& y = \arg\min \|V_{j+1} r_0 - Hy\|_2 \\
& x = x_0 + V_j y \\
& if\ \|Ax - b\|_2 < tolerance \quad Stop \\
& else\ x_0 = x \ \ and\ goto\ start
\end{aligned}
$$

Figure 2.2 GMRES Algorithm

3. DOMAIN DECOMPOSITION SOLVER

Domain decomposition [1-2, 4, 11, 16] algorithm is a powerful method for solving large scale system of equations arising from discretization of partial differential equations (PDE) in finite element procedures. The computational domain is decomposed into smaller sub-domains each of which is easier to solve.

Domain decomposition (DD) is an application of the divide-and-conquer problem-solving strategy, which consists of expressing a large problem as a set of smaller sub-problems defined on sub-domains and provides a way to determine the solution of the original problem in terms of solutions to sub-problems.

The goal of DD is to divide the original problem into sub-problems that can be solved independently. A critical issue in DD is to assure that the sub-problems preserve the solution to the original problem.

Let us assume the system of linear algebraic equations

$$Kr = f \tag{3.1}$$

where the matrix of system is $K \in R^{n \times n}$ and vectors $r \in R^n, f \in R^n$. It is possible to split Eq. (3.1) into blocks (or sub-matrices)

$$\begin{bmatrix} K_{11} & K_{12} \\ K_{21} & K_{22} \end{bmatrix} \begin{bmatrix} r^{(1)} \\ r^{(2)} \end{bmatrix} = \begin{bmatrix} f^{(1)} \\ f^{(2)} \end{bmatrix} \tag{3.2}$$

Eq. (3.2) can be written as

$$K_{11} r^{(1)} + K_{12} r^{(2)} = f^{(1)} \tag{3.3}$$

$$K_{21} r^{(1)} + K_{22} r^{(2)} = f^{(2)} \tag{3.4}$$

From Eq. (3.3), the vector $r^{(1)}$ can be expressed in the form

$$r^{(1)} = K_{11}^{-1}\left(f^{(1)} - K_{12}r^{(2)}\right) \tag{3.5}$$

Substituting Eq. (3.5) in Eq. (3.4)

$$(K_{22} - K_{21}K_{11}^{-1}K_{12})r^{(2)} = f^{(2)} - K_{21}K_{11}^{-1}f^{(1)} \tag{3.6}$$

From Eq. (3.6) we can observe that the number of unknowns have been reduced as this matrix equation is only related to the unknown vector $r^{(2)}$.

The matrix $(K_{22} - K_{21}K_{11}^{-1}K_{12})$ is often referred as the Schur's complement in the mathematical community.

The basic ideas in DD solver is to solve for the unknown vector $\{r\}$, shown in Eq. (3.1), in the following 2 major steps:

Step 1: The "boundary unknown" vector $\{r^{(2)}\}$ is solved from Eq. (3.6). Since the triple matrix products appeared in Eq. (3.6) is usually dense, and computationally expensive (because K_{12} is a matrix, and not a vector), iterative solver (which is based on matrix times vector operations) is usually recommended in this step.

Step 2: The "interior unknown" vector $\{r^{(1)}\}$ is solved from Eq. (3.5). Since the coefficient matrix $[K_{11}]$ is usually sparse (and $f^{(1)}$ is a vector, not a matrix), direct solver (such as Cholesky, or $LDL^T, or\ LU$ algorithms) is strongly recommended.

4. GENERALIZED INVERSE ALGORITHMS FOR SINGULAR/NON-SINGULAR, SQUARE/RECTANGULAR SYSTEM OF SLE

In Chapter 2, various direct and iterative methods for solving square/non-singular system of SLE have been summarized. The (direct and iterative) methods described in Chapter 2 can be significantly enhanced/improved by the Domain Decomposition (DD) formulation [1-2, 11, 16], described in Chapter 3. Domain decomposition formulation has been widely adopted, since it can take full advantage of "parallel processing" capability offered by most (if not all) today super-computers, and even to desktop/laptop computers, which have multiple processors. In this Chapter 4, however, the main focus is shifted into algorithms (numerical procedures) that can solve much more general classes of SLE, shown in Eq. (2.1), for which the coefficient matrix $[A]$ can be square/rectangular, non-singular/singular, well-posed/illed condition. These desirable algorithms have been based on the so called Generalized (or Pseudo, or Moore-Penrose) inverse of a general matrix [5-9, 11-13, 17-18]. Furthermore, DD formulation will also be used in this chapter to further improve the numerical performance of the generalized inverse.

4.1 Basic Conditions for the Generalized Inverse

The Moore-Penrose inverse (or generalized inverse or pseudo inverse) of a $m \times n$ matrix K (not necessarily a square matrix) is the unique $n \times m$ matrix K^+ which satisfies the following four conditions:

1. General condition: $KK^+K = K$,

2. Reflexive condition: $K^+KK^+ = K^+$,

3. Normalized condition: $(KK^+)' = KK^+$,

4. Reverse normalized condition: $(K^+K)' = K^+K$

4.2 Potential Engineering/Science Generalized Inverse Applications

There are some applications that result in (or lead to) an inconsistent system of SLE. A solution may not exist in this case. However, we can consider to fit a vector $\vec{x}$ to a given inconsistent system. That is, we can define a least-squares error problem for finding $\vec{x}$ that minimizes the absolute error $\|A\vec{x} - \vec{b}\|_2$ which is equivalent to minimizing the summation of the square of the error $\|A\vec{x} - \vec{b}\|_2^2$. As should be obvious, such a solution vector $\vec{x}$ may not be unique, and it can be obtained/computed by $\vec{x} = A^+\vec{b}$, where $A^+$is generalized (or pseudo-) inverse of A.

It has been well documented in the literature that many real-life engineering/science/statistical applications can be efficiently solved by efficient computation of the generalized inverse in conjunction with SLE. In the following section, it can be shown that the popular "least square problem" can be formulated such that generalized inverse algorithms can be incorporated for obtaining the desired solution.

4.3 Least Squares Curve Fitting Problem

Let us assume we are given a set of data points $(x_1, y_1), (x_2, y_2), \dots (x_n, y_n)$, and we wish to find parameters $c_1, c_2, \dots c_k$ to be multiplied by the basis functions $\emptyset_1, \emptyset_2, .. \emptyset_k$ such that the scalar function S, defined as

$$S = \sum_{i=1}^{n} |\varepsilon_i|^2 \tag{4.1}$$

is minimized, where the errors ε_i can be computed by

$$\begin{aligned}
\varepsilon_1 &= c_1\phi_1(x_1)+c_2\phi_2(x_1)+.......+c_k\phi_k(x_1)-y_1\\
\varepsilon_2 &= c_1\phi_1(x_2)+c_2\phi_2(x_2)+.......+c_k\phi_k(x_2)-y_2\\
&\vdots \qquad\qquad\qquad \vdots \qquad\qquad\qquad \vdots\\
\varepsilon_n &= c_1\phi_1(x_n)+c_2\phi_2(x_n)+.......+c_k\phi_k(x_n)-y_n
\end{aligned} \tag{4.2}$$

In Eq. (4.1), the scalar parameter S represents the "summation of the square of the errors". Ideally, we would like to set Eq. (4.1) to zero, i.e., each of the ε_i to be zero. Otherwise, we would like to minimize the scalar function S. If c *and* y are the vectors

$$\begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_k \end{bmatrix} \text{ and } \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \tag{4.3}$$

respectively, and A is the $n \times k$ matrix

$$A = \begin{bmatrix} \phi_1(x_1) & \phi_2(x_1) & \cdots & \phi_k(x_1) \\ \phi_1(x_2) & \phi_2(x_2) & \cdots & \phi_k(x_2) \\ \vdots & \vdots & \vdots & \vdots \\ \phi_1(x_n) & \phi_2(x_n) & \cdots & \phi_k(x_n) \end{bmatrix} \tag{4.4}$$

Then, the Right-Hand-Side (or RHS) of Eq. (4.2) can also be written in the form

$$\begin{pmatrix} \phi_1(x_1) & \phi_2(x_1) & \cdots & \phi_k(x_1) \\ \phi_1(x_2) & \phi_2(x_2) & \cdots & \phi_k(x_2) \\ \vdots & \vdots & \vdots & \vdots \\ \phi_1(x_n) & \phi_2(x_n) & \cdots & \phi_k(x_n) \end{pmatrix}_{n\times k} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_k \end{pmatrix}_{k\times 1} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}_{n\times 1} \tag{4.5}$$

The requirement $S = 0$ is equivalent to

$$Ac = y \tag{4.6}$$

From Eq. (4.5), if $n > k$, we have an over-determined system of linear equations, since the number of equations is larger than that of the unknowns. It is generally not possible to find a solution to this system, but we may find $c_1, c_2, ..c_k$ such that Ac is close to y (in the least squares sense). If there is a solution c^+ of the least squares problem, then we write

$$c^+ = A^+ y \tag{4.7}$$

The matrix A^+ is called the pseudo-inverse (or generalized inverse) of A. We know that when $n = k$ and A is invertible (i.e., A has an inverse), then

$$A^+ = A^{-1} \tag{4.8}$$

4.3.1 The Normal Equations [3, 10, 19]

In order to minimize S, we must minimize

$$S_{\min} \equiv \sum_{i=1}^{n} [c_1\phi_1(x_i) + c_2\phi_2(x_i) + \dots\dots + c_k\phi_k(x_i) - y_i]^2 \tag{4.9}$$

If we take for $j = 1,2,\dots,k$ the partial derivatives of S with respect to c_j and set them equal to zero, we will get the following normal equations

$$2\sum_{i=1}^{n} [c_1\phi_1(x_i) + c_2\phi_2(x_i) + \dots\dots + c_k\phi_k(x_i) - y_i]\phi_j(x_i) = 0 \tag{4.10}$$

or, equivalently,

$$\sum_{i=1}^{n} [c_1\phi_1(x_i) + c_2\phi_2(x_i) + \dots\dots + c_k\phi_k(x_i)]\phi_j(x_i) = \sum_{i=1}^{n} \phi_j(x_i) y_i \tag{4.11}$$

The above normal equations can also be expressed (in matrix notations) as

$$A^T A c = A^T y \tag{4.12}$$

where the matrix A has the entries $a_{ij} = \phi_j(x_i)$.

As an example, let $k = 4$ and $\phi_j(x) = x^{j-1}$, where $j = 1,2,\dots,k = 4$

Then,

$$\phi_1(x) = 1$$
$$\phi_2(x) = x$$
$$\phi_3(x) = x^2$$
$$\phi_4(x) = x^3$$

and

$$A = \begin{pmatrix} 1 & x_1 & x_1^2 & x_1^3 \\ 1 & x_2 & x_2^2 & x_2^3 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_n & x_n^2 & x_n^3 \end{pmatrix} \tag{4.13}$$

$$A^T = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_n \\ x_1^2 & x_2^2 & \cdots & x_n^2 \\ x_1^3 & x_2^3 & \cdots & x_n^3 \end{pmatrix} \tag{4.14}$$

so that

$$A^T A = \begin{pmatrix} n & \sum x_i & \sum x_i^2 & \sum x_i^3 \\ \sum x_i & \sum x_i^2 & \sum x_i^3 & \sum x_i^4 \\ \sum x_i^2 & \sum x_i^3 & \sum x_i^4 & \sum x_i^5 \\ \sum x_i^3 & \sum x_i^4 & \sum x_i^5 & \sum x_i^6 \end{pmatrix} \tag{4.15}$$

where in the summations, i runs from 1 *to* n and

$$A^T y = \begin{pmatrix} \sum y_i \\ \sum x_i y_i \\ \sum x_i^2 y_i \\ \sum x_i^3 y_i \end{pmatrix} \tag{4.16}$$

If the matrix $A^T A$ in Eq. (4.15) is invertible, then there is a solution to the least squares problem.

We then have

$$c = \left(A^T A\right)^{-1} A^T y, \tag{4.17}$$

so that the pseudo-inverse A^+ (of the given matrix A, see Eqs. 4.6-4.7) is given by

$$A^+ = \left(A^T A\right)^{-1} A^T \tag{4.18}$$

In subsequent sections, Eq. (4.18) can be generalized into the form shown in Eq. (4.27).

4.4 Special Cholesky Algorithms for Factorizing a Singular Matrix

The stiffness matrix of the "floating" sub-domain is singular due to the fact that there are not enough (support) constraints to prevent its rigid body motion. To facilitate the discussions, let's assume the "floating" sub-domain's stiffness matrix is given as:

$$\left[K_{float}\right] = \begin{bmatrix} 1 & 2 & -3 & 2 & -2 & -3 & -2 \\ 2 & 4 & -6 & 4 & -4 & -6 & -4 \\ -3 & -6 & 9 & -6 & 6 & 9 & 6 \\ 2 & 4 & -6 & 5 & -1 & -5 & -7 \\ -2 & -4 & 6 & -1 & 13 & 9 & -5 \\ -3 & -6 & 9 & -5 & 9 & 13 & 9 \\ -2 & -4 & 6 & -7 & -5 & 9 & 27 \end{bmatrix} \tag{4.19}$$

$$row2 = 2 * row1 \tag{4.20}$$

$$row3 = -3 * row1 \tag{4.21}$$

$$row5 = -8 * row1 + 3 * row4 \tag{4.22}$$

In Eq. (4.19), it can be observed that rows number 2, 3 and 5 are dependent rows (and columns). Thus, the above matrix is singular, and the regular/standard Cholesky factorization algorithms will not work. To facilitate the development of efficient "generalized inverse" algorithms (and its applications) in subsequent chapters, "special

Cholesky" factorization algorithm is needed. The "special Cholesky factorization" algorithm is essentially identical to the regular/standard one, with the following two modifications (or differences):

(a) During the numerical factorization phase, if the dependent row(s) is/are detected, then these dependent row(s) is/are skipped!

(b) Factorization of the current i^{th} row, in general, will require the previously already factorized rows $k = 1,2,\ldots,i-1$. However, if the previous k^{th}row was amongst the dependent row(s), then the contribution from this k^{th}row to this current i^{th} row will be ignored.

(c) Having obtained the "special Cholesky" factorized square matrix, those factorized/dependent rows will be deleted to obtain the so-called "truncated Cholesky factorized rectangular matrix U^*".

Using MATLAB, the Eigen-values of matrix in Eq. (4.19) can be computed as:

$$\vec{\lambda} = \{0.0000, \quad 0.0000, \quad 0.0000, \quad 0.2372, \quad 4.9375, \quad 24.9641, \quad 41.8612\}^T$$

Since there are 3 zero Eigen-values, it implies there are 3 rigid body modes (or 3 dependent rows/columns) in Eq. (4.19)

If row-by-row Cholesky factorization scheme is applied to Eq. (4.19), we will encounter that the factorized $u_{22} = 0 = u_{33} = u_{55}$, which indicated that row number 2, 3 and 5 are dependent rows. Thus, if we set all factorized terms of rows number 2, 3 and 5 are zero, and "ignoring" these three rows in the factorization of subsequent rows, one obtains the following Cholesky factorized matrix $[U]$ of a given matrix $[K_{float}]$, shown in Eq. (4.19):

$$[U] = \begin{bmatrix} 1 & 2 & -3 & 2 & -2 & -3 & -2 \\ . & 0 & 0 & 0 & 0 & 0 & 0 \\ . & . & 0 & 0 & 0 & 0 & 0 \\ . & . & . & 1 & 3 & 1 & -3 \\ . & . & . & . & 0 & 0 & 0 \\ . & . & . & . & . & 1.7321 & 3.464 \\ . & . & . & . & . & . & 1.4145 \end{bmatrix}$$

Based on the above "special Cholesky factorization" algorithm, and using the 7×7 singular matrix data as shown in Eq. (4.19), the "truncated Cholesky factorized matrix U^*" (corresponding to the independent rows # 1, 4, 6, and 7) of the product $[K_{float}]^T * [K_{float}]$ can be obtained/computed as:

For row #1 of the truncated 4×7 matrix U^*:

$5.9160 \quad 11.8321 \quad -17.7482 \quad 11.6631 \quad -12.3392 \quad -21.4669 \quad -19.1004$

For row #4 of the truncated 4×7 matrix U^*:

$0.0000 \quad 0.0000 \quad 0.0000 \quad 4.4689 \quad 13.4068 \quad 0.9781 \quad -21.7565$

For row #6 of the truncated 4×7 matrix U^*:

$0.0000 \quad 0.0000 \quad 0.0000 \quad 0.0000 \quad 0.0000 \quad 4.4960 \quad 10.0650$

For row #7 of the truncated 4×7 matrix U^*:

$0.0000 \quad 0.0000 \quad 0.0000 \quad 0.0000 \quad 0.0000 \quad 0.0000 \quad 0.7209$

4.5 Special LDL^T Algorithms for Factorizing a Singular Matrix

If the square/symmetrical/singular coefficient matrix is NOT positive definite, then the symmetrical, floating stiffness matrix (shown in Eq. 4.19) can be factorized by the familiar sparse LDL^T algorithms [11], with the following minor modifications:

(a) Whenever a dependent i^{th} row is encountered (such as the factorized $u_{ii} = 0$), then the following things need be done:

(a.1) Recording the dependent row number(s). For the data given by Eq. (4.19), the dependent rows are rows number 2, 3 and 5.

(a.2) Set all the nonzero terms of the factorized i^{th} row (of L^T) to be zero

(a.3) Set $\frac{1}{u_{ii}} \equiv D_{ii} = 0$

(b) Whenever an independent i^{th} row is encountered, the factorized i^{th} row will have contributions from all appropriated previously factorized rows. However, contributions from the previously factorized (dependent) rows will be ignored.

(c) Finally, the truncated/rectangular LDL^T factorized matrix U^* can be obtained by deleting those dependent row(s) from the "special LDL^T factorized" square matrix.

As an example, the (LDL^T) factorized matrix (for the matrix data $[K_{float}]$ shown in Eq. (4.19)) can be computed as

$$[U] = \begin{bmatrix} 1 & 2 & -3 & 2 & -2 & -3 & -2 \\ . & 0 & 0 & 0 & 0 & 0 & 0 \\ . & . & 0 & 0 & 0 & 0 & 0 \\ . & . & . & 1 & 3 & 1 & -3 \\ . & . & . & . & 0 & 0 & 0 \\ . & . & . & . & . & 0.333 & 2 \\ . & . & . & . & . & . & 0.5 \end{bmatrix} \tag{4.23}$$

and the truncated factorized LDL^T matrix U^* of the product $[K_{float}]^T * [K_{float}]$ can be obtained by deleting rows #2, 3, and 5 of the above matrix U.

A complete educational version of "special LDL^T " software code (written in FORTRAN language) is listed/given in the Appendix B.

4.6 Efficient Generalized Inverse Algorithms

Moore-Penrose inverse can be computed using Singular Value Decomposition (SVD) [10-13], Least Squares Method, QR factorizations, Finite Recursive Algorithm [9], etc. In this work, our numerical algorithms have been based on:

(a) The "special Cholesky factorization" (for symmetrical/singular coefficient matrix) see section 4.4 and the Appendix C, and

(b) The generalized inverse of a product of 2 matrices [5, 13]

and can be described in the following paragraphs.

Consider Eq. (2.1) $[G]\vec{x} = \vec{b}$, with a square coefficient $n \times n$ matrix, and let the rank be less than the size of the matrix (if r is the rank of the matrix, then $r \leq n$). Let the size of the known right-hand-side vector $\vec{b}$ be $n \times 1$. Consider a symmetric positive $n \times n$ matrix $G'G$, with rank $r \leq n$ (here, the matrix $[G]$ plays the same role as matrix $[A]$ in Eq. (2.1)), then based on the theorem presented in [5, 13, 17-18], there exists a unique $[M]$ such that:

$$G'G = M'M \tag{4.24}$$

In Eq. (4.24), matrices $[G']$ *and* $[G]$ have the dimensions $n \times m$ *and* $m \times n$, respectively.

M is the upper triangular (special) Cholesky factorized matrix and contains exactly $n-r$ zero rows. Removing the zero rows from M, one obtains a $r \times n$ (upper, rectangular) matrix L'.

$$A \equiv M'M = LL' \tag{4.25}$$

In this work, the upper triangular (special) Cholesky factorized matrix $[M]$ can be obtained by the regular/standard Cholesky factorization, with the following modifications:

a) When the diagonal term of the current i^{th} row is very close to zero, then factorization of this dependent row is skipped.

b) When the current i^{th} row is factorized, all previous rows $k = 1,2,\ldots,i-1$ were used except those dependent row(s).

Consider the generalized inverse of a matrix product AB [5, 13, 17-18]

$$(AB)^{+} = B'(A'ABB')^{+}A' \tag{4.26}$$

From Eq. (4.26), if $B = I$ then

$$A^{+} = (A'A)^{+}A' \tag{4.27}$$

Eq. (4.27) can be considered as a general version of the earlier Eq. (4.18).

If $B = A'$ and A is a $n \times r$ matrix of rank r, then one obtains from Eq. (4.26)

$$(AA')^{+} = (A')'\left(A'AA'(A')'\right)^{+}A' \tag{4.28}$$

Let us consider regular inverse in Eq. (4.28) in place of generalized inverse

$$(AA')^{+} = A(A'AA'A)^{-1}A'$$

$$= A(A'A)^{-1}(A'A)^{-1}A' \tag{4.29}$$

Using Eq. (4.27),

$$G^{+} = (G'G)^{+}G' \tag{4.30}$$

From Eqs. (4.24 – 4.25) and Eq. (4.29) one obtains,

$$(G'G)^{+} = (LL')^{+} = L(L'L)^{-1}(L'L)^{-1}L' \tag{4.31}$$

Thus, Eq. (4.30) becomes

$$G^{+} = (G'G)^{+} G' = L(L'L)^{-1}(L'L)^{-1} L'G' \tag{4.32}$$

While MATLAB solution can be obtained by $\vec{x} = pinv(G) \times \vec{b}$, implying the generalized inverse G^{+} [see Eq. 4.32] to be formed explicitly, our main idea is to solve SLE where $\vec{b}$ is a known right-hand-side vector, as described in the next section.

To facilitate the discussion of Generalized Inverse, and its usage for solving general systems of SLE, the following (small-scale) numerical examples are used:

Example 4.1

The coefficient matrix $[G]$ is a rectangular (tall type) matrix, and the RHS vector $\{b\}$ is a linear combinations of columns of $[G]$.

In this example, we wish to solve for $\{x\}$ from the SLE $[G] * \{x\} = \{b\}$, where the numerical values of the coefficient matrix $[G]$, and the RHS vector $\{b\}$ are given as [refer to Eq. (4.32)]:

$$\begin{bmatrix} 2 & 0 & 0 & 1 & 1 \\ -1 & 0 & 0 & -2 & -1 \\ 3 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & -3 & 1 \\ 0 & -2 & 5 & 1 & 0 \\ 1 & -2 & 3 & 4 & -1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{Bmatrix} = \begin{bmatrix} 6 \\ -5 \\ 6 \\ 1 \\ 4 \\ 6 \end{bmatrix}$$

Using MATLAB built-in function, the rank of G can be computed as

$rank(G) = 5$

and $G^T * G = \begin{bmatrix} -15 & -2 & 3 & 8 & 2 \\ -2 & 9 & -14 & -13 & 3 \\ 3 & -14 & 38 & 11 & -1 \\ 8 & -13 & 11 & 31 & -4 \\ 2 & 3 & -1 & -4 & 4 \end{bmatrix}$

Special Cholesky factor of $G^T * G$

$$= \begin{bmatrix} 3.8729 & -0.5163 & 0.7745 & 2.0655 & 0.5163 \\ 0.0000 & 2.9552 & -4.6020 & -4.0380 & 1.1053 \\ 0.0000 & 0.0000 & 4.0275 & -2.2800 & 0.9154 \\ 0.0000 & 0.0000 & 0.0000 & 2.2866 & 0.64909 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 1.1189 \end{bmatrix}$$

# of independent rows = 5

Independent rows = 1 2 3 4 5

Dependent row = 6

Product of $L^T * L$

$$= \begin{bmatrix} 20.4000 & -12.8609 & -1.1172 & 5.0584 & 0.5778 \\ -12.8609 & 47.4396 & -8.3159 & -8.5160 & 1.2368 \\ -1.1172 & -8.3159 & 22.2581 & -4.6195 & 1.0243 \\ 5.0584 & -8.5160 & -4.6195 & 5.6500 & 0.7263 \\ 0.5778 & 1.2368 & 1.0243 & 0.7263 & 1.2520 \end{bmatrix}$$

Regular Cholesky factorization of $L^T * L$

$$= \begin{bmatrix} 4.5166 & -2.8474 & -0.2473 & 1.1199 & 0.1279 \\ 0.0000 & 6.2714 & -1.4383 & -0.8494 & 0.2553 \\ 0.0000 & 0.0000 & 4.4864 & -1.2402 & 0.3172 \\ 0.0000 & 0.0000 & 0.0000 & 1.4615 & 0.8164 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.6350 \end{bmatrix}$$

Generalized inverse of $[G]$

$$= \begin{bmatrix} 0.07957 & 0.0648 & 0.2930 & 0.0117 & -0.0206 & 0.0265 \\ -0.4289 & -0.3494 & -0.1162 & 0.9364 & -0.8888 & 0.8570 \\ -0.1342 & -0.0723 & -0.0308 & 0.3504 & -0.1133 & 0.2885 \\ -0.0645 & -0.2377 & -0.1401 & 0.1385 & -0.2425 & 0.3118 \\ 0.4337 & -0.2761 & -0.2072 & -0.2320 & 0.4060 & -0.5220 \end{bmatrix}$$

and the solution vector $\vec{x}$ is obtained as

$$\begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{Bmatrix} = \begin{Bmatrix} 2.0000 \\ 0.9999 \\ 0.9999 \\ 0.9999 \\ 1.0000 \end{Bmatrix}$$

Example 4.2

The coefficient matrix $[G]$ is a square/singular matrix, and the RHS vector $\{b\}$ is a random vector.

$$\begin{bmatrix} 2 & -1 & 0 & 0 & 1 & -1 \\ 4 & -2 & 0 & 0 & 2 & -2 \\ 0 & 0 & 1 & 2 & 0 & 2 \\ 0 & 0 & -2 & 5 & -1 & 1 \\ 1 & 3 & 0 & 1 & 1 & 3 \\ 1 & 1 & 0 & 2 & 5 & 7 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{Bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 7 \\ 4 \\ 12 \\ 23 \end{bmatrix}$$

Using MATLAB built-in function, the rank of G can be computed as

$rank(G) = 5$

$$G^T * G = \begin{bmatrix} 22 & -6 & 0 & 3 & 16 & 0 \\ -6 & 15 & 0 & 5 & 3 & 21 \\ 0 & 0 & 5 & -8 & 2 & 0 \\ 3 & 5 & -8 & 34 & 6 & 26 \\ 16 & 3 & 2 & 6 & 32 & 32 \\ 0 & 21 & 0 & 26 & 32 & 68 \end{bmatrix}$$

Special Cholesky factor of $G^T * G$

$$= \begin{bmatrix} 4.6904 & -1.2792 & 0.0000 & 0.6396 & 0.3411 & 0.0000 \\ 0.0000 & 3.6556 & 0.0000 & 1.5915 & 2.0143 & 5.7445 \\ 0.0000 & 0.0000 & 2.2360 & -3.5777 & 0.8944 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 4.2729 & 0.8921 & 3.9451 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 3.8353 & 4.4086 \end{bmatrix}$$

# of independent rows = 5

Independent rows = 1 2 3 4 5

Dependent row = 6

Product of $L^T * L$

$$= \begin{bmatrix} 35.6818 & 3.2129 & 0.7627 & 5.7764 & 13.0832 \\ 3.2129 & 52.9542 & -3.8924 & 31.2607 & 33.0513 \\ 0.7627 & -3.8924 & 18.6000 & -14.4892 & 3.4304 \\ 5.7764 & 31.2607 & -14.4892 & 34.6177 & 20.8144 \\ 13.0832 & 33.0513 & 3.4304 & 20.8144 & 34.1462 \end{bmatrix}$$

Regular Cholesky factorization of $L^T * L$

$$= \begin{bmatrix} 5.9734 & 0.5378 & 0.1276 & 0.9670 & 2.1902 \\ 0.0000 & 7.2570 & -0.5458 & 4.2359 & 4.3920 \\ 0.0000 & 0.0000 & 4.2761 & -2.8765 & 1.2974 \\ 0.0000 & 0.0000 & 0.0000 & 2.7321 & 1.3996 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 2.5331 \end{bmatrix}$$

Generalized inverse of $[G]$

$$= \begin{bmatrix} 0.0749 & 0.1498 & 0.0823 & -0.0161 & 0.1882 & -0.0742 \\ -0.0107 & -0.0215 & -0.1076 & -5.146 \times 10^{-3} & 0.3118 & -0.0879 \\ 0.0182 & 0.0364 & 0.5157 & -0.1939 & 0.0305 & -0.0980 \\ 0.0151 & 0.0302 & 0.1518 & 0.1278 & -7.108 \times 10^{-4} & -0.0271 \\ 0.0150 & 0.0301 & -0.1820 & -3.678 \times 10^{-3} & -0.0793 & 0.1366 \\ -0.0242 & -0.0485 & 0.0902 & -0.0308 & -0.0145 & 0.0761 \end{bmatrix}$$

and the solution vector $\vec{x}$ is obtained as

$$\begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{Bmatrix} = \begin{Bmatrix} 1.0638 \\ 0.9459 \\ 0.9465 \\ 0.9423 \\ 0.9029 \\ 2.0845 \end{Bmatrix}$$

Example 4.3

The coefficient matrix $[G]$ is a square/non-singular matrix, and the RHS vector $\{b\}$ is a linear combination of columns of $[G]$.

$$\begin{bmatrix} 2 & -1 & 0 & 0 & 1 & -1 \\ 4 & -2 & 0 & 0 & -2 & -2 \\ 0 & 0 & 1 & 2 & 0 & 2 \\ 0 & 0 & -2 & 5 & -1 & 1 \\ 1 & 3 & 0 & 1 & 1 & 3 \\ 1 & 3 & 0 & 1 & 6 & 1 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{Bmatrix} = \begin{bmatrix} 1 \\ -2 \\ 5 \\ 3 \\ 9 \\ 12 \end{bmatrix}$$

Using MATLAB built-in function, the rank of G can be computed as

$rank(G) = 6$

$$G^T * G = \begin{bmatrix} 22 & -4 & 0 & 2 & 1 & -6 \\ -4 & 23 & 0 & 6 & 24 & 17 \\ 0 & 0 & 5 & -8 & 2 & 0 \\ 2 & 6 & -8 & 31 & 2 & 13 \\ 1 & 24 & 2 & 2 & 43 & 11 \\ -6 & 17 & 0 & 13 & 11 & 20 \end{bmatrix}$$

Special Cholesky factor of $G^T * G$

$$= \begin{bmatrix} 4.6904 & -0.8528 & 0.0000 & 0.4204 & 0.2132 & -1.2792 \\ 0.0000 & 4.7193 & 0.0000 & 1.3483 & 5.1239 & 3.3709 \\ 0.0000 & 0.0000 & 2.2360 & -3.5777 & 0.8944 & 0.0000 \\ 0.0000 & 0.0000 & 0.0000 & 4.0249 & -0.4472 & 2.2360 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 3.9623 & -1.2618 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.6384 \end{bmatrix}$$

# of independent rows = 6

Independent rows = 1 2 3 4 5 6

Dependent row = none

Product of $L^T * L$

$$= \begin{bmatrix} 24.5909 & -6.6695 & -1.3348 & -1.2395 & 2.4589 & -0.8167 \\ -6.6695 & 61.7090 & -0.2412 & 10.6735 & 16.0488 & 2.1522 \\ -1.3348 & -0.2412 & 18.6000 & -14.8000 & 3.5440 & 0.0000 \\ -1.2395 & 10.6735 & -14.8000 & 21.4000 & -4.5936 & 1.4276 \\ 2.4589 & 16.0488 & 3.5440 & -4.5936 & 17.2923 & -0.8056 \\ -0.8167 & 2.1522 & 0.0000 & 1.4276 & -0.8056 & 0.4076 \end{bmatrix}$$

Regular Cholesky factorization of $L^T * L$

$$= \begin{bmatrix} 4.9589 & -1.3449 & -0.2691 & -0.2499 & 0.4958 & -0.1646 \\ 0.0000 & 7.7390 & -0.0779 & 1.3356 & 2.1597 & 0.2494 \\ 0.0000 & 0.0000 & 4.3036 & -3.4303 & 0.8936 & -5.78 \times 10^{-3} \\ 0.0000 & 0.0000 & 0.0000 & 2.7903 & -1.5370 & 0.3703 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 3.0365 & -0.2266 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.36012 \end{bmatrix}$$

Generalized inverse of $[G]$

$$
= \begin{bmatrix} 0.2341 & 0.0972 & -0.0317 & -0.0158 & 0.1825 & -0.0396 \\ -1.2817 & 0.5694 & -0.0634 & -0.0317 & -0.1349 & 0.4206 \\ -1.2222 & 0.6111 & 0.5555 & -0.2222 & -0.4444 & -0.4444 \\ -0.6388 & 0.3194 & 0.2222 & 0.1111 & -0.2777 & 0.2777 \\ 0.5000 & -0.2500 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ 1.2500 & -0.6250 & 0.0000 & 0.0000 & 0.5000 & -0.5000 \end{bmatrix}
$$

and the solution vector $\vec{x}$ is obtained as

$$
\begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{Bmatrix} = \begin{Bmatrix} 1.0000 \\ 0.9999 \\ 0.9999 \\ 0.9999 \\ 1.0000 \\ 1.0000 \end{Bmatrix}
$$

Example 4.4

The coefficient matrix $[G]$ is a square/singular matrix, and the RHS vector $\{b\}$ is a random vector.

$$
\begin{bmatrix} 2 & -1 & 0 & 0 & 0 & 1 & -1 \\ 4 & 2 & 0 & 0 & 0 & 2 & -2 \\ 0 & 0 & 1 & 2 & -5 & -3 & 6 \\ 0 & 0 & -2 & 5 & 6 & 2 & 1 \\ 0 & 0 & 3 & 5 & 1 & 7 & 0 \\ 1 & 2 & -3 & 6 & -2 & 1 & 3 \\ 2 & 4 & -6 & 12 & -4 & 2 & 6 \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{Bmatrix} = \begin{bmatrix} 1 \\ 6 \\ 5 \\ 3 \\ 9 \\ 9 \\ 9 \end{bmatrix}
$$

Using MATLAB built-in function, the rank of G can be computed as

$rank(G) = 6$

$$
G^T * G = \begin{bmatrix} 25 & 16 & -15 & 30 & -10 & 15 & 5 \\ 16 & 25 & -30 & 60 & -20 & 13 & 27 \\ -15 & -30 & 59 & -83 & 16 & -1 & -41 \\ 30 & 60 & -83 & 234 & -35 & 69 & 107 \\ -10 & -20 & 16 & -35 & 82 & 24 & -54 \\ -15 & 13 & -1 & 69 & 24 & 72 & -6 \\ 5 & 27 & -41 & 107 & -54 & -6 & 87 \end{bmatrix}
$$

Special Cholesky factor of $G^T * G$

$$
= \begin{bmatrix} 5.0000 & 3.2000 & -3.0000 & 6.0000 & -2.0000 & 3.0000 & 1.0000 \\ 0.0000 & 3.8418 & -5.3099 & 10.6198 & -3.5399 & 0.8849 & -1.0933 \\ 0.0000 & 0.0000 & 4.6695 & -1.8438 & -1.8838 & 2.7195 & -1.0933 \\ 0.0000 & 0.0000 & 0.0000 & 9.0454 & 1.2293 & 5.1535 & 3.6698 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 7.7722 & 4.1069 & -4.7144 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 3.3756 & -3.2763 \end{bmatrix}
$$

# of independent rows = 6

Independent rows = 1 2 3 4 5 6

Dependent row = 7

Product of $L^T * L$

$$
= \begin{bmatrix} 94.2400 & 107.8723 & -14.2385 & 70.9443 & -7.9381 & 6.8505 \\ 107.8723 & 207.4266 & -42.0738 & 119.0041 & -53.0841 & -17.3091 \\ -14.2385 & -42.0738 & 37.3448 & -8.9911 & 1.6820 & 12.7625 \\ 70.9443 & 119.0041 & -8.9911 & 123.3579 & 13.4190 & 5.3727 \\ -7.9381 & -53.0841 & 1.6820 & 13.4190 & 99.5613 & 29.3094 \\ 6.8505 & -17.3091 & 12.7625 & 5.3727 & 29.3094 & 22.1291 \end{bmatrix}
$$

Regular Cholesky factorization of $L^T * L$

$$
= \begin{bmatrix} 9.7077 & 11.1120 & -1.4667 & 7.3080 & -0.8177 & 0.7056 \\ 0.0000 & 9.1624 & -2.8131 & 4.1252 & -4.8019 & -2.7449 \\ 0.0000 & 0.0000 & 5.2229 & 2.55271 & -2.4940 & 1.1632 \\ 0.0000 & 0.0000 & 0.0000 & 0.81297 & 6.6882 & 1.2578 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 4.9813 & 2.2468 \\ 0.0000 & 0.0000 & 0.0000 & 0.0000 & 0.0000 & 2.4723 \end{bmatrix}
$$

Generalized inverse of $[G]$

$$
= \begin{bmatrix} 0.2050 & 0.1624 & 0.1051 & 0.0748 & -0.0440 & -0.0119 & -0.0239 \\ -0.4999 & 0.2500 & 0.0000 & -1.0547 & 0.0000 & 0.0000 & 0.0000 \\ -0.1361 & 0.1144 & 0.1337 & 0.0393 & 0.0869 & -0.0364 & -0.0729 \\ 0.0579 & -0.0468 & 2.017 \times 10^{-4} & 0.0254 & 0.0272 & 0.0151 & 0.0303 \\ -0.0938 & 0.0780 & 0.0410 & 0.1627 & -0.0238 & -0.0249 & -0.0499 \\ 0.0303 & -0.0267 & -0.0633 & -0.0582 & 0.0895 & 8.3858 \times 10^{-3} & 0.0167 \\ -0.0596 & 0.0482 & 0.1468 & 0.0914 & 1.3269 \times 10^{-3} & -0.0156 & -0.0312 \end{bmatrix}
$$

and the solution vector $\vec{x}$ is obtained from Eq. (4.32) as

$$\begin{Bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{Bmatrix} = \begin{Bmatrix} 1.2092 \\ 1.0000 \\ 1.1341 \\ 0.5089 \\ 0.1789 \\ 0.4105 \\ 0.8290 \end{Bmatrix}$$

4.7 Mixed Direct-Iterative Generalized Inverse Algorithms for Solving SLE

Instead of explicitly computing the generalized inverse from Eq. (4.32), which involves a lot of "matrix times matrix" operations, one can/should repeatedly use "matrix times vector" operations for improving its computational efficiency. Furthermore, it is noted that the "regular (not generalized) inverse" of the matrix product $[L' * L]$ should be replaced by the more efficient SLE, either by Direct or by Iterative algorithms. More details can be explained in the following sub-sections.

4.7.1 Direct methods in Generalized Inverse for solving SLE

Using the matrix-product operations in Eq. (4.32), one can compute the unknown solution vector $\vec{x}$ according to the following sequence of steps:

From Eq. (2.1), with $[A] \equiv [G]$, and from Eq. (4.32),

$$\vec{x} = L(L'L)^{-1}(L'L)^{-1}L'G'\vec{b}$$

Let

$$Minv = (L'L)^{-1}$$

Then one computes the following sequence of matrix times vector:

$$tempo1 = G' \times \vec{b}$$

$$tempo3 = L' \times tempo1$$

$tempo2 = Minv \times tempo3$

$tempo1 = Minv \times tempo2$

$tempo2 = L \times tempo1$

and the unknown solution vector $\vec{x}$ is stored in the temporary vector $tempo2$.

4.7.2 Iterative Method in Generalized Inverse for solving SLE

From Eq. (4.32),

$$\vec{x} = L(L'L)^{-1}(L'L)^{-1}L'G'\vec{b}$$

Let

$$M1 = (L'L)$$

Then one computes the following sequence of matrix times vector:

$tempo1 = G' \times \vec{b}$

$tempo3 = L' \times tempo1$

Using MATLAB built-in (Conjugate Gradient) function, one computes the following vectors:

$tempo2 = cg(M1, tempo3)$

$tempo1 = cg(M1, tempo2)$

$tempo2 = L \times tempo1$

and the unknown solution vector $\vec{x}$ is stored in the temporary vector $tempo2$.

where cg is the Conjugate Gradient (iterative) Algorithm for solving SLE and the unknown solution vector $\vec{x}$ is stored in the temporary vector $tempo2$.

Important Notes:

(a) Inside the "generalized inverse" algorithms, one needs to find the "regular inverse" of the coefficient matrix ($= L' * L$, in this case). This "regular matrix inversion" should be equivalently solved by SLE, which can be done either by direct, or iterative solvers.

(b) For certain large-scale (especially for 3-D) problems, iterative solver (with appropriated pre-conditioned strategies) can be a more preferable method of choice (as compared to the direct method), due to the following desirable features:

- No fill-in terms occurred
- Much easier to parallelize

4.8 Domain Decomposition Generalized Inverse Solver

The efficient generalized inverse algorithms discussed in sections 4.6 and 4.7 can be further improved by utilizing the Domain Decomposition (or DD) formulation. In Chapter 3, some key equations resulted from DD formulation have already been derived, based on the assumption that the coefficient of a square matrix is non-singular.

In this section, let us consider the system of linear algebraic equations

$$Kr = f \tag{4.33}$$

where the matrix of system is $K \in R^{n \times n}$ (K can be either non-singular, or singular) and vectors $r \in R^n$, $f \in R^n$.

Let the original domain be decomposed into m sub domains. The coefficient matrix can be represented in a special form as shown below

$$\begin{bmatrix} K_1^{[ii]} & & O & & & K_1^{[ib]} \\ & K_2^{[ii]} & & & & K_2^{[ib]} \\ O & & K_3^{[ii]} & & & K_3^{[ib]} \\ \vdots & & & \ddots & & \vdots \\ & & & & K_m^{[ii]} & K_m^{[ib]} \\ K_1^{[bi]} & K_2^{[bi]} & K_3^{[bi]} & \cdots & K_m^{[bi]} & K^{[bb]} \end{bmatrix} \begin{pmatrix} r_{[i]}^{(1)} \\ r_{[i]}^{(2)} \\ r_{[i]}^{(3)} \\ \vdots \\ r_{[i]}^{(m)} \\ r_{[b]} \end{pmatrix} = \begin{pmatrix} f_{[i]}^{(1)} \\ f_{[i]}^{(2)} \\ f_{[i]}^{(3)} \\ \vdots \\ f_{[i]}^{(m)} \\ f_{[b]} \end{pmatrix} \tag{4.34}$$

From Eq. (4.34) the following notations are used

$r_{[i]}^{[j]}$, *where* $j \in \{1,2,...,m\}$ - vectors of displacements of interior nodes on the j^{th} sub domain,

$r_{[b]}$ - vector of boundary displacements,

$f_{[i]}^{[j]}$, *where* $j \in \{1,2,...,m\}$ - vectors of interior loads acting on j^{th} sub domain,

$f_{[b]}$ - vector of boundary loads on the boundaries.

$K^{[bb]}$ represents the summation of boundary stiffness matrices from all sub-domains.

For "structural engineering" applications, and using DD formulation, Eq. (4.33) can be naturally expressed in the form as shown in Eq. (4.34). For general "field" problems (or mathematical problems), one usually starts with Eq. (4.33). For these problems, however, Nested Disection or METiS re-ordering algorithms [20-22] can be used to transform Eq. (4.33) into the form shown in Eq. (4.34).

From Eq. (4.34), matrix $K^{[bb]}$ contains contributions from "all" sub-domains and can be expressed as

$$K^{[bb]} = \sum_{j=1}^{m} K_j^{[bb]} \tag{4.35}$$

For symmetrical cases, $K_j^{[ib]}$ *and* $K_j^{[bi]}$ are the transpose of each other. Vectors $r_{[i]}^{[j]}$ can be expressed in the form

$$r_{[i]}^{[j]} = \left(K_j^{[ii]}\right)^{-1}\left(f_i^{[j]} - K_j^{[ib]} r_b^{[j]}\right) \tag{4.36}$$

where $j \in \{1,2,\ldots,m\}$, *and* $r_b^{[j]}$ is a sub-set of the vector $r_{[b]}$.

Also,

$$\left(K^{[bb]} - \sum_{j=1}^{m} K_j^{[bi]}\left(K_j^{[ii]}\right)^{-1} K_j^{[ib]}\right) r_{[b]} = f_b - \sum_{j=1}^{m} K_j^{[bi]}\left(K_j^{[ii]}\right)^{-1} f_i^{[j]} \tag{4.37}$$

Eq. (4.37) is called the reduced system or resulting system.

If the matrix K, shown in Eq. (4.33), is singular, we can apply the concept of generalized inverse to invert the (possible) singular coefficient matrix in Eq. (4.37) to find the unknown boundary displacement vector $r_{[b]}$ first, and then use them in computing the remaining unknown interior displacement vector $r_{[i]}^{[j]}$, as shown in Eq. (4.36).

For a typical j^{th} sub-domain $j \in \{1,2,\ldots,m\}$, one obtains

$$r_{[I]}^{(j)} = \left[K_j^{(II)}\right]^{-1} \cdot \left(f_I^{(j)} - K_j^{(IB)} r_{[b]}^{[j]}\right) \tag{4.38}$$

5. ENGINEERING/SCIENCES NUMERICAL APPLICATIONS

Based on the discussions presented in the previous chapters (chapters 1-4), a fairly extensive set of numerical examples are used in this chapter for validating the accuracy and evaluating the (computational time) performance of the proposed algorithms discussed in Chapter 4. Test examples to cover the cases where the known/given coefficient matrix in the SLE can be a square/rectangular, singular/non-singular, symmetrical/non-symmetrical matrix, and the known/given right-hand-side (rhs) vector can be random vector, or it can be a linear combinations of columns of the coefficient matrix are all investigated in this chapter.

5.1 Description of the Test Examples

Test matrices are collected from Tim Davis Sparse Matrix Collection [14], University of Florida. Rank deficient (singular) matrices derived from various engineering and science applications such as Linear Programming, Combinatorial Problem, Directed Graph, Fluid Dynamics, Linear Programming, Chemical Process Simulations, Cell traffic matrices, etc. are included in Tables 5.1-5.4

Sl. No	*Name*	*Size*	*Rank*	*nnz*	*Group*	*Description*
1	lock_700	700x700	165	22,175	HB	Finite Element Problem
2	dwt_1005	1005x1005	995	8,621	HB	Structural Problem
3	bcspwr06	1454x1454	1446	5,300	HB	Power network problem
4	bcsstm13	2003x2003	1241	21,181	HB	Symmetric Mass Matrix, Fluid Flow Generalized Eigen Values
5	lock2232	2232x2232	368	80,352	HB	Finite Element Problem
6	cegb2802	2802x2802	289	277,362	HB	Finite Element Problem

Table 5.1: Symmetric Singular Test Matrices for ODU Generalized Inverse Solver

Sl. No	*Name*	*Size*	*Rank*	*nnz*	*Group*	*Description*
1	tomo_900	900x900	893	35,598	Regtools	2-D tomography test problem
2	GD00_c	638x638	300	1,041	Pajek network	Directed Multigraph
3	GD96_a	1096x1096	827	1,677	Pajek network	Directed Multigraph
4	ex_6	1651x1651	1650	49,062	FIDAP	CFD
5	CS_phd	1882x1882	705	1,740	Pajek network	Directed Graph
6	tomo_2500	2500x2500	2496	166,782	Regtools	2-D tomography test problem

Table 5.2: Un-symmetrical Singular Test Matrices for ODU Generalized Inverse Solver

Sl. No	*Name*	*Size*	*Rank*	*nnz*	*Group*	*Description*
1	D_6	970x435	339	6,491	JGD_SL6	Differentials of Voronoi complex of perfect forms
2	mk9-b2	1260x378	343	3,780	JGD_Homology	Combinatorial problem
3	n3c6-b3	1365x455	364	5,460	JGD_Homology	Combinatorial problem
4	Franz1	2240x768	755	5,120	JGD_Franz	Combinatorial problem
5	mk10-b2	3150x630	586	9,450	JGD_Homology	Simplical complexes
6	n4c6-b3	5970x1330	1140	23,880	JGD_Homology	Simplical complexes from Homology from Volkmar Welker

Table 5.3: Rectangular Singular Test Matrices (Tall type: rows>>cols) for ODU Generalized Inverse Solver

Sl. No	*Name*	*Size*	*Rank*	*nnz*	*Group*	*Description*
1	lp_standgub	361x1383	360	3,338	LPnetlib	Linear Programming Problem
2	lp_ship04l	402x2166	360	6,380	LPnetlib	Linear Programming Problem
3	lp_ship08s	778x2467	712	7,194	LPnetlib	Linear Programming Problem
4	Trec12	551x2726	550	151,219	JGD_Kocay	combinatorial problem
5	lp_ship08l	778x4363	712	12,882	LPnetlib	Linear Programming Problem
6	lp_d6cube	415x6184	404	37,704	LPnetlib	Linear Programming Problem

Table 5.4: Rectangular Singular Test Matrices (Fat type: rows<<cols) for ODU Generalized Inverse Solver

5.2 Numerical Performance of ODU Generalized Inverse Solver

Based on the detailed algorithms explained in Chapter 4, and using the rank-deficient matrices as coefficient matrices described in section 5.1, the numerical performance of our proposed procedures [for solving SLE, shown in Eq. (4.33)] are evaluated in this section. The known RHS vector $\{b\}$ can be random vector, or can be chosen such a way that the unknown solution vector $\{x\} = \{1, 1, \dots, 1\}$.

We also compared the performance of our algorithm with the efficient algorithm described in [13] and also with MATLAB built-in function *pinv*() [12] for computing the generalized inverse explicitly. We use MATLAB version 7.6.0.324 (R2008a) on Intel Core 2 CPU, 2.13GHZ, 2GB RAM, Windows XP Professional SP3 for numerical comparisons. To be consistent and fair, sparse test matrices obtained from tables 5.1-5.4 are converted to full matrices (in this section).

Tables 5.5 through 5.16 records the times (in seconds) taken by our proposed algorithm, the algorithm mentioned in [13] and MATLAB built-in function [12] *pinv*(). For our convenience, we represent our algorithm with $ODU - ginverse()$, algorithm in [13] with $geninv$ and MATLAB built-in function with $MATLAB - pinv()$. In addition, we have also presented the error norm for all the test matrices.

5.2.1 Direct Method in Generalized Inverse to Solve SLE with RHS Vector as Linear Combination of Columns of the Coefficient Matrix

In this sub-section, the explicitly inverse of the matrix product$[L' * L]$, shown in Eq. (4.32), is implemented by MATLAB built-in function $inv(L' * L)$, and the results are shown in Tables 5.5-5.12.

Sl. No	*Name*	*Size*	*Rank*	$ODU - ginverse$ *Error Norm*	$geninv$ *Error Norm*	$MATLAB - pinv()$ *Error Norm*
1	lock_700	700x700	165	0.1514 1.033×10^{-8}	0.3446 1.1399×10^{-6}	1.2967 2.215×10^{-11}
2	dwt_1005	1005x1005	995	2.6634 7.1302×10^{-9}	4.2889 6.764×10^{-6}	14.0320 4.5736×10^{-12}
3	bcspwr06	1454x1454	1446	8.5029 1.477×10^{-8}	13.3176 1.829×10^{-5}	40.3646 2.7131×10^{-12}
4	bcsstm13	2003x2003	1241	11.5997 6.7629×10^{-9}	19.1901 1.826×10^{-8}	36.3413 5.6493×10^{-13}
5	lock2232	2232x2232	368	5.5518 7.9519×10^{-9}	10.8755 2.5797×10^{-7}	40.7582 1.0761×10^{-11}
6	cegb2802	2802x2802	289	8.9571 9.7558×10^{-9}	18.6816 3.7220×10^{-7}	69.9847 1.7532×10^{-11}

Table 5.5: Computational Times (in seconds) for Symmetric Rank-Deficient Test Matrices with RHS Vector as Linear Combination of Columns of Coefficient Matrix

Sl. No	Name	Size	Rank	ODU – ginverse Error Norm	geninv Error Norm	MATLAB – pinv() Error Norm
1	tomo_900	900x900	893	1.8545 7.667×10^{-9}	3.0092 1.055×10^{-6}	10.1396 2.789×10^{-11}
2	GD00_c	638x638	300	0.1805 7.909×10^{-12}	0.3961 2.602×10^{-11}	1.8696 5.704×10^{-13}
3	GD96_a	1096x1096	827	2.5991 3.179×10^{-13}	4.1606 1.776×10^{-13}	7.3928 3.148×10^{-13}
4	ex_6	1651x1651	1650	12.78109 2.547×10^{-5}	19.9238 0.00657	44.6059 4.6022×10^{-12}
5	CS_phd	1882x1882	705	8.5161 7.724×10^{-14}	12.8672 9.295×10^{-14}	41.8010 2.707×10^{-13}
6	tomo_2500	2500x2500	2496	44.7203 2.031×10^{-7}	69.0133 0.0002893	221.6490 2.8190×10^{-10}

Table 5.6: Computational Times (in seconds) for Non-Symmetric Rank-Deficient Test Matrices with RHS Vector as Linear Combination of Columns of Coefficient Matrix

Sl. No	Name	Size	Rank	ODU – ginverse Error Norm	geninv Error Norm	MATLAB – pinv() Error Norm
1	D_6	970x435	339	0.1347 1.216×10^{-11}	0.2809 1.333×10^{-11}	1.3240 7.403×10^{-13}
2	mk9-b2	1260x378	343	0.1162 5.950×10^{-14}	0.2478 1.018×10^{-13}	0.6098 1.681×10^{-13}
3	Franz1	2240x768	755	1.3077 1.457×10^{-13}	2.3649 1.290×10^{-13}	6.0490 2.806×10^{-13}
4	mk10-b2	3150x630	586	0.8094 1.599×10^{-13}	1.5776 2.057×10^{-13}	3.2363 2.573×10^{-13}

Table 5.7: Computational Times (in seconds) for Rectangular Rank-Deficient Test Matrices (Tall type: Rows>>Cols) with RHS Vector as Linear Combination of Columns of Coefficient Matrix

Sl. No	Name	Size	Rank	ODU – ginverse Error Norm	geninv Error Norm	MATLAB – pinv() Error Norm
1	lp_standgub	361x1383	360	0.1242 6.321×10^{-7}	0.4238 6.387×10^{-7}	1.0215 3.579×10^{-11}
2	lp_ship04l	402x2166	360	0.1760 1.421×10^{-11}	0.6712 1.056×10^{-11}	1.3390 1.851×10^{-12}
3	lp_ship08s	778x2467	712	1.2747 1.178×10^{-11}	3.4073 1.052×10^{-11}	5.3489 1.583×10^{-12}
4	lp_ship08l	778x4363	712	1.5622 2.955×10^{-11}	5.2861 1.517×10^{-11}	8.5278 4.243×10^{-12}

Table 5.8: Computational Times (in seconds) for Rectangular Rank-Deficient Test Matrices (Fat type: Rows<<Cols) with RHS Vector as Linear Combination of Columns of Coefficient Matrix

5.2.2 Direct Method in Generalized Inverse to Solve SLE with Randomly Generated RHS Vector $\{1, 2, \dots, n\}^T$.

Sl. No	*Name*	*Size*	*Rank*	*ODU – ginverse* *Error Norm*	*geninv* *Error Norm*	*MATLAB – pinv()* *Error Norm*
1	lock_700	700x700	165	0.1483 *1495.46*	0.3423 *1495.46*	1.2948 *1495.46*
2	dwt_1005	1005x1005	995	2.6740 *295*	4.3156 *295*	14.0342 *295*
3	bcspwr06	1454x1454	1446	8.5747 *102.551*	13.3752 *102.551*	41.1197 *102.551*
4	bcsstm13	2003x2003	1241	11.7270 *29034.7*	19.3159 *29034.7*	36.3659 *29034.7*
5	lock2232	2232x2232	368	5.5934 *106.489*	10.9423 *106.489*	40.7723 *106.489*
6	cegb2802	2802x2802	289	9.1066 *28002*	18.7714 *28002*	69.7151 *28002*

Table 5.9: Computational Times (in seconds) for Symmetric Rank-Deficient Test Matrices with Randomly Generated RHS Vector

Sl. No	*Name*	*Size*	*Rank*	*ODU – ginverse* *Error Norm*	*geninv* *Error Norm*	*MATLAB – pinv()* *Error Norm*
1	tomo_900	900x900	893	1.8725 *669.871*	3.0287 *669.871*	10.0959 *669.871*
2	GD00_c	638x638	300	0.2012 *6204.66*	0.4092 *6204.66*	1.8748 *6204.66*
3	GD96_a	1096x1096	827	2.6079 *4821.7*	4.1399 *4821.7*	7.3863 *4821.7*
4	ex_6	1651x1651	1650	12.7676 *1682.04*	20.0046 *1682.04*	44.5628 *1682.04*
5	CS_phd	1882x1882	705	8.5641 *40498.9*	12.9633 *40498.9*	41.7883 *40498.9*
6	tomo_2500	2500x2500	2496	45.0147 *1561.25*	69.2692 *1561.25*	221.4820 *1561.25*

Table 5.10: Computational Times (in seconds) for Non-Symmetric Rank-Deficient Test Matrices with Randomly Generated RHS Vector

Sl. No	Name	Size	Rank	ODU – ginverse Error Norm	geninv Error Norm	MATLAB – pinv() Error Norm
1	D_6	970x435	339	0.1991 *14795*	0.3480 *14795*	1.3203 *14795*
2	mk9-b2	1260x378	343	0.1615 *6376.16*	0.2935 *6376.16*	0.6090 *6376.16*
3	Franzl	2240x768	755	1.2964 *37127.5*	2.3602 *37127.5*	6.0413 *37127.5*
4	mk10-b2	3150x630	586	0.8063 *26222.6*	1.5845 *26222.6*	3.2379 *26222.6*

Table 5.11: Computational Times (in seconds) for Rectangular Rank-Deficient Test Matrices (Tall type: Rows>>Cols) with Randomly Generated RHS Vector

Sl. No	Name	Size	Rank	ODU – ginverse Error Norm	geninv Error Norm	MATLAB – pinv() Error Norm
1	lp_standgub	361x1383	360	0.1586 *2*	0.4580 *2*	1.0321 *2*
2	lp_ship04l	402x2166	360	0.2250 *1508.7*	0.7126 *1508.7*	1.3383 *1508.7*
3	lp_ship08s	778x2467	712	1.2945 *833468*	3.3961 *833468*	5.3447 *833468*
5	lp_ship08l	778x4363	712	1.5722 *3724.81*	5.2922 *3724.81*	8.5757 *3724.81*

Table 5.12: Computational Times (in seconds) for Rectangular Rank-Deficient Test Matrices (Fat type: Rows<<Cols) with Randomly Generated RHS Vector

5.2.3 Iterative Methods in Generalized Inverse to Solve SLE with Randomly Generated RHS vector

In this sub-section, the explicit inverse of the matrix product $[L' * L]$, shown in Eq. (4.32), is implemented by MATLAB built-in (Conjugate Gradient iterative solver), and the results are shown in Tables 5.13-5.16.

Iterative solver used: Conjugate Gradient Algorithm

Error tolerance used: 10^{-7}

Sl. No	*Name*	*Size*	*Rank*	*ODU – ginverse iterative* *Error Norm*	*ODU – ginverse direct* *Error Norm*	*geninv()* *Error Norm*	*MATLAB pinv()* *Error Norm*
1	lock1074	1074x1074	155	0.5649 *47.5347*	0.4351 *47.5347*	1.0278 *47.5347*	4.1070 *47.5347*
2	lock2232	2232x2232	368	5.9314 *63.1189*	5.5499 *63.1189*	10.8729 *63.1189*	40.7251 *63.1189*
3	cegb2802	2802x2802	289	9.3092 *77.0117*	9.0585 *77.0117*	18.6983 *77.0117*	69.7249 *77.0117*

Table 5.13: Computational Times (in seconds) for Symmetric Rank-Deficient Test Matrices (Using Iterative Solver inside Generalized Inverse) with Randomly Generated RHS Vector

Sl. No	*Name*	*Size*	*Rank*	*ODU – ginverse iterative* *Error Norm*	*ODU – ginverse direct* *Error Norm*	*geninv()* *Error Norm*	*MATLAB pinv()* *Error Norm*
1	GD00_c	638x638	300	0.2835 *46.1318*	0.2125 *46.1318*	0.4097 *46.1318*	1.8861 *46.1318*
2	GD96_a	1096x1096	827	2.7595 *28.0286*	2.6291 *28.0286*	4.1385 *28.0286*	7.3832 *28.0286*
3	CS_phd	1882x1882	705	8.5563 *96.906*	8.5541 *96.906*	12.9385 *96.906*	41.6813 *96.906*

Table 5.14: Computational Times (in seconds) for Non-Symmetric Rank-Deficient Test Matrices (Using Iterative Solver inside Generalized Inverse) with Randomly Generated RHS Vector

Sl. No	*Name*	*Size*	*Rank*	*ODU – ginverse iterative* *Error Norm*	*ODU – ginverse direct* *Error Norm*	*geninv()* *Error Norm*	*MATLAB pinv()* *Error Norm*
1	mk9-b2	1260x378	343	0.1423 *43.7972*	0.1615 *43.7972*	0.2928 *43.7972*	0.6096 *43.7972*
2	Franzl	2240x768	755	1.2200 *84.9266*	1.3051 *84.9266*	2.3673 *84.9266*	6.0349 *84.9266*
3	n4c6-b3	5970x1330	1140	7.26591 *128.086*	7.8267 *128.086*	14.1278 *128.086*	25.7394 *128.086*

Table 5.15: Computational Times (in seconds) for Rectangular (Tall) Rank-Deficient Test Matrices (Using Iterative Solver inside Generalized Inverse) with Randomly Generated RHS Vector

Sl. No	*Name*	*Size*	*Rank*	*ODU − ginverse iterative* *Error Norm*	*ODU − ginverse direct* *Error Norm*	*geninv()* *Error Norm*	*MATLAB pinv(* *Error Norm*
1	lp_ship08s	778x2467	712	1.59103 *21.3141*	1.2930 *21.3141*	3.3817 *21.3141*	5.3262 *21.3141*
2	lp_ship08l	778x4363	712	1.7496 *24.7184*	1.6065 *24.7184*	5.2864 *24.7184*	8.4837 *24.7184*
3	lp_d6cube	415x6184	404	0.5738 *10.7908*	0.46281 *10.7908*	2.1500 *10.7908*	4.1469 *10.7908*

Table 5.16: Computational Times (in seconds) for Rectangular (Fat) Rank-Deficient Test Matrices (Using Iterative Solver inside Generalized Inverse) with Randomly Generated RHS Vector

Furthermore, graphical comparisons (in terms of the computational times) of ODU-ginverse with other algorithms have been presented in Appendix E.

5.3 Numerical Performance of ODU Generalized Inverse DD Solver

As can be seen from the previous Section 5.2, our proposed implementation of the generalized inverse for solving SLE have shown "significant time reduction" as compared to MATLAB built-in function, and also compared to [13]. In this sub-section, however, we want to investigate the numerical performance of our proposed generalized inverse within the frame work of DD formulation (for solving system of $[G] * \{x\} = \{b\}$, where the coefficient matrix $[G]$ has the dimension $m \times n$ and can be either rectangular, or square/singular).

5.3.1 Description of Test Problems for ODU Generalized Inverse DD Solver

Test matrices are collected from Tim Davis Sparse Matrix Collection, University of Florida [14]. Rank deficient (singular) matrices derived from various engineering and science applications such as Linear Programming, Combinatorial Problem, Directed Graph, Fluid Dynamics, Linear Programming, Chemical Process Simulations, Cell traffic Matrices, etc. are included in Tables 5.17-5.18

Sl. No	Name	Size	Rank	nnz	Group	Description/Kind
1	GD98_c	112x112	100	336	Pajek	Pajek network, undirected graph
2	dwt_209	209x209	208	1,743	HB	Structural Problem
3	dwt_307	307x307	288	2,523	HB	Structural Problem

Table 5.17: Symmetric Singular Test Matrices for ODU Generalized Inverse DD Solver

Sl. No	Name	Size	Rank	nnz	Group	Description/Kind
1	gent113	113x113	107	655	HB	Statistical/Mathematical Problem
2	gre_216b	216x216	215	812	HB	Directed Weight Graph
3	GD00_a	352x352	178	458	Pajek	Directed Graph

Table 5.18: Un-Symmetrical Singular Test Matrices for ODU Generalized Inverse DD Solver

5.3.2 Problem Formulation with Same Sub Matrix

Let us construct a system matrix (Eq. (4.33)) in the form shown by Eq. (4.34). For the sake of discussion, let us consider 3 sub domains. The system matrix takes the following form

$$\begin{pmatrix} K_1^{ii} & O & O & K_1^{ib} \\ O & K_2^{ii} & O & K_2^{ib} \\ O & O & K_3^{ii} & K_3^{ib} \\ K_1^{bi} & K_2^{bi} & K_3^{bi} & K^{bb} \end{pmatrix} \begin{pmatrix} r_i^1 \\ r_i^2 \\ r_i^3 \\ r_b \end{pmatrix} = \begin{pmatrix} f_i^1 \\ f_i^2 \\ f_i^3 \\ f_b \end{pmatrix} \tag{5.1}$$

A singular coefficient matrix is considered from the test matrix collection [14-15] as K_1^{ii}. For simplicity/convenience, we also assume

(a) $K_1^{ii} = K_2^{ii} = K_3^{ii} = K^{bb}$ (5.2)

(b) $K_1^{ib} = K_2^{ib} = K_3^{ib} = K_1^{bi} = K_2^{bi} = K_3^{bi} = K_1^{ii}$ (5.3)

The known right hand side (RHS) vector is chosen as $factor \times random\, column\, vector$. Where *factor* is a user defined variable. The user can also specify the number of sub domains and, accordingly, the system matrix and the right hand side vector will be automatically generated.

This section presents a comparison of numerical results (in terms of timings) of the developed Domain Decomposition generalized inverse formulations with other existing algorithms. MATLAB sparse storage scheme has been adopted on the input test matrices.

Sl. No	*nsu*	*Size*	*Rank*	*ODU – DD* *Error Norm*	*Original system ginverse MV* *Error Norm*	*Original system geninv* *Error Norm*
1	2	336x336	300	0.06130 *8.21375*	0.07158 *8.21375*	0.11555 *8.21375*
2	3	448x448	400	0.09911 *12.2835*	0.15793 *12.2835*	0.26940 *12.2835*
3	4	560x560	500	0.12202 *9.95631*	0.30984 *9.95631*	0.49831 *9.95631*
4	5	672x672	600	0.15667 *15.2519*	0.56595 *15.2519*	0.88289 *15.2519*
5	6	784x784	700	0.2000 *13.1714*	0.95323 *13.1714*	1.45982 *13.1714*
6	7	896x896	800	0.20218 *14.9267*	1.44812 *14.9267*	2.19521 *14.9267*
7	8	1008x1008	900	0.24897 *14.6983*	2.16118 *14.6983*	3.21151 *14.6983*

Table 5.19: Computational Times (in seconds) for Rank-Deficient Test Matrices with same K^{ii} (GD98_c) Sub-Matrices using Domain Decomposition

Sl. No	*nsu*	*Size*	*Rank*	*ODU – DD* *Error Norm*	*Original system ginverse MV* *Error Norm*	*Original system geninv* *Error Norm*
1	2	627x627	624	0.28611 *2.9162*	0.53233 *2.9162*	0.859960 *2.9162*
2	3	836x836	832	0.401712 *3.87701*	1.37505 *3.87701*	2.10186 *3.87701*
3	4	1045x1045	1040	0.53614 *2.0488*	2.98979 *2.0488*	4.44309 *2.0488*
4	5	1254x1254	1248	0.59997 *2.54757*	5.08382 *2.54757*	7.43006 *2.54757*
5	6	1463x1463	1456	0.71609 *1.06024*	8.2171 *1.06024*	12.06235 *1.06024*
6	7	1672x1672	1664	0.87530 *2.14487*	12.46184 *2.14487*	18.24484 *2.14487*
7	8	1881x1881	1872	1.00417 *8.99101*	18.2003 *8.99101*	25.81453 *8.99101*

Table 5.20: Computational Times (in seconds) for Rank-Deficient Test Matrices with same K^{ii} (dwt_209) Sub-Matrices using Domain Decomposition

Sl. No	*nsu*	*Size*	*Rank*	*ODU – DD* Error Norm	*Original system ginverse MV* Error Norm	*Original system geninv* Error Norm
1	2	921x921	864	0.67732 *14.6443*	1.70853 *14.6443*	2.59315 *14.6443*
2	3	1228x1228	1152	1.0100 *12.991*	4.38649 *12.991*	6.43760 *12.991*
3	4	1535x1535	1440	1.2845 *11.2669*	8.6928 *11.2669*	12.76263 *11.2669*
4	5	1842x1842	1728	1.62886 *17.1461*	15.2775 *17.1461*	22.0046 *17.1461*
5	6	2149x2149	2016	1.96057 *17.6758*	26.66948 *17.6758*	35.4461 *17.6758*
6	7	2456x2456	2304	2.28013 *17.0915*	36.77451 *17.0915*	53.29277 *17.0915*
7	8	2763x2763	2592	2.70171 *21.1027*	52.6385 *21.1027*	75.09115 *21.1027*

Table 5.21: Computational Times (in seconds) for Rank-Deficient Test Matrices with same K^{ii} (dwt_307) Sub-Matrices using Domain Decomposition

Sl. No	*nsu*	*Size*	*Rank*	*ODU – DD* Error Norm	*Original system ginverse MV* Error Norm	*Original system geninv* Error Norm
1	2	339x339	321	0.075614 *4.83871*	0.093916 *4.83871*	0.14541 *4.83871*
2	3	452x452	428	0.010403 *7.01463*	0.19469 *7.01463*	0.305175 *7.01463*
3	4	565x565	535	0.134215 *9.03996*	0.35666 *9.03996*	0.575419 *9.03996*
4	5	678x678	642	0.166451 *10.8548*	0.65779 *10.8548*	1.03111 *10.8548*
5	6	791x791	749	0.20053 *10.1139*	1.31634 *10.1139*	1.89611 *10.1139*
6	7	904x904	856	0.224461 *10.1037*	1.64580 *10.1037*	2.46109 *10.1037*
7	8	1017x1017	963	0.30215 *8.71228*	2.433860 *8.71228*	3.65811 *8.71228*

Table 5.22: Computational Times (in seconds) for Rank-Deficient Test Matrices with same K^{ii} (gent113) Sub-Matrices using Domain Decomposition

Sl. No	*nsu*	*Size*	*Rank*	*ODU – DD* Error Norm	*Original system ginverse MV* Error Norm	*Original system geninv* Error Norm
1	2	648x648	643	0.213104 *29.2826*	0.48045 *29.2826*	0.73973 *29.2826*
2	3	864x864	857	0.32912 *32.058*	1.17688 *32.058*	1.77665 *32.058*
3	4	1080x1080	1066	0.45151 *36.0931*	2.500164 *36.0931*	3.57136 *36.0931*
4	5	1296x1296	1279	0.53808 *40.1448*	4.33999 *40.1448*	6.28897 *40.1448*
5	6	1512x1512	1492	0.64925 *45.0203*	7.19551 *45.0203*	10.12725 *45.0203*
6	7	1728x1728	1705	0.699806 *44.2327*	10.70643 *44.2327*	15.20850 *44.2327*
7	8	1944x1944	1918	0.80099 *54.8367*	16.44834 *54.8367*	23.81176 *54.8367*

Table 5.23: Computational Times (in seconds) for Rank-Deficient Test Matrices with same K^{ii} (gre_216b) Sub-Matrices using Domain Decomposition

Sl. No	*nsu*	*Size*	*Rank*	*ODU – DD* Error Norm	*Original system ginverse MV* Error Norm	*Original system geninv* Error Norm
1	2	1056x1056	534	0.38620 *50.4652*	1.05101 *50.4652*	1.62067 *50.4652*
2	3	1408x1408	712	0.56582 *56.6697*	2.96705 *56.6697*	4.05349 *56.6697*
3	4	1760x1760	890	0.71932 *63.9613*	5.83588 *63.9613*	8.01557 *63.9613*
4	5	2112x2112	1068	0.888563 *69.849*	10.48973 *69.849*	14.15703 *69.849*
5	6	2464x2464	1246	1.16060 *79.2452*	16.4323 *79.2452*	22.84216 *79.2452*
6	7	2816x2816	1424	1.48376 *82.7129*	25.37213 *82.7129*	34.37000 *82.7129*
7	8	3168x3168	1602	1.69688 *85.8676*	36.3547 *85.8676*	49.16511 *85.8676*

Table 5.24: Computational Times (in seconds) for Rank-Deficient Test Matrices with same K^{ii} (GD00_a) Sub-Matrices using Domain Decomposition

5.3.3 Problem Formulation with Different Sub Matrix

Let us construct a system matrix (Eq. (4.33)) in the form shown by Eq. (4.34). For the sake of discussion, let us consider 3 sub domains. The system matrix takes of the form

$$\begin{pmatrix} K_1^{ii} & O & O & K_1^{ib} \\ O & K_2^{ii} & O & K_2^{ib} \\ O & O & K_3^{ii} & K_3^{ib} \\ K_1^{bi} & K_2^{bi} & K_3^{bi} & K^{bb} \end{pmatrix} \begin{pmatrix} r_i^1 \\ r_i^2 \\ r_i^3 \\ r_b \end{pmatrix} = \begin{pmatrix} f_i^1 \\ f_i^2 \\ f_i^3 \\ f_b \end{pmatrix}$$

A singular coefficient matrix is considered from the test matrix collection [14-15] as K_1^{ii}. For simplicity, we also assume

(a) $K_1^{ii} \neq K_2^{ii} \neq K_3^{ii}$

(b) $K_1^{ii} = K_1^{ib} = K_1^{bi}$

(c) $K_2^{ii} = K_2^{ib} = K_2^{bi}$

(d) $K_3^{ii} = K_3^{ib} = K_3^{bi}$

(e) $K_1^{ii} = K^{bb}$

The known right hand side vector is chosen as $f_i = 1{:}m, f_b = 1{:}m \; where \; [m,n] = size(K_1^{ii})$. The user can also specify the number of sub domains and accordingly, the system matrix and the right hand side vector will be generated.

Numerical results for these test cases are presented in Tables 5.25-5.27

Sl. No	*nsu*	*Size*	*Rank*	*ODU – DD* Error Norm	*Original system ginverse MV* Error Norm	*Original system geninv* Error Norm
1	9	1610x1610	1573	0.59484 *12.8452*	10.75132 *12.8452*	15.65746 *12.8452*
2	10	1771x1771	1725	0.65966 *15.7321*	14.52799 *15.7321*	21.00736 *15.7321*
3	11	1932x1932	1876	0.71032 *18.9077*	19.01395 *18.9077*	26.94307 *18.9077*

Table 5.25: Computational Times (in seconds) for Rank-Deficient Test Matrices with different K^{ii} (can_161) Sub-Matrices using Domain Decomposition

Sl. No	*nsu*	*Size*	*Rank*	*ODU – DD* Error Norm	*Original system ginverse MV* Error Norm	*Original system geninv* Error Norm
1	3	2800x2800	660	2.46978 *2990.92*	10.66420 *2990.92*	14.53973 *2990.92*
2	4	3500x3500	825	2.83254 *3343.95*	22.08674 *3343.95*	29.95532 *3343.95*
3	5	4200x4200	990	3.88394 *3663.12*	37.47156 *3663.12*	49.44602 *3663.12*

Table 5.26: Computational Times (in seconds) for Rank-Deficient Test Matrices with different K^{ii} (lock_700) Sub-Matrices using Domain Decomposition

Sl. No	*nsu*	*Size*	*Rank*	*ODU – DD* Error Norm	*Original system ginverse MV* Error Norm	*Original system geninv* Error Norm
1	4	1445x1445	1438	1.23158 *2.73861*	7.89617 *2.73861*	11.5578 *2.73861*
2	5	1734x1734	1723	1.33245 *4.1833*	14.00956 *4.1833*	20.0745 *4.1833*
3	6	2023x2023	2007	1.55488 *5.91608*	22.07078 *5.91608*	31.8324 *5.91608*

Table 5.27: Computational Times (in seconds) for Rank-Deficient Test Matrices with different K^{ii} (mesh3e1) Sub-Matrices using Domain Decomposition

5.3.4 Problem Formulation with Same Sub-Matrices and RHS as Linear Combinations of Columns

The known right hand side vector is chosen as $C_1 + C_2 + C_3 + \cdots + C_n$, where n is the number of columns, and C_i (with $i = 1,2,\ldots,n$) represents the i^{th} column of the coefficient matrix $[G]$ in the big matrix.

Numerical results for the test cases are presented in Tables 5.28-5.30

Sl. No	nsu	Size	Rank	ODU – DD $\sum \lvert x_i \rvert$ Error Norm	Original system ginverse MV $\sum \lvert x_i \rvert$ Error Norm	Original system geninv $\sum \lvert x_i \rvert$ Error Norm
1	3	2800x2800	660	2.52323 2764 1.9738×10^{-6}	10.26388 2764 5.0396×10^{-7}	14.11214 2764 0.00016343
2	4	3500x3500	825	3.31814 3455 2.2792×10^{-6}	20.38195 3455 3.0938×10^{-7}	27.63699 3455 0.0001999
3	5	4200x4200	990	4.04986 4146 2.5482×10^{-6}	35.52977 4146 7.7514×10^{-7}	48.13018 4146 0.0002406

Table 5.28: Computational Times (in seconds) for Rank-Deficient Test Matrices with same K^{ii} (lock_700) Sub-Matrices and RHS as Linear Combinations of Columns $C_1 + C_2 + C_3 + \cdots + C_n$ using Domain Decomposition

Sl. No	nsu	Size	Rank	ODU – DD $\sum \lvert x_i \rvert$ Error Norm	Original system ginverse MV $\sum \lvert x_i \rvert$ Error Norm	Original system geninv $\sum \lvert x_i \rvert$ Error Norm
1	3	1228x1228	1152	1.05252 1228 3.5175×10^{-7}	4.20519 1228 2.2430×10^{-8}	6.20720 1228 2.4071×10^{-5}
2	4	1535x1535	1440	1.37372 1535 4.0617×10^{-7}	8.58372 1535 5.7408×10^{-8}	12.5739 1535 3.1755×10^{-5}
3	5	1842x1842	1728	1.70161 1842 4.5413×10^{-7}	14.99398 1842 4.0396×10^{-8}	21.9261 1842 4.3923×10^{-5}

Table 5.29: Computational Times (in seconds) for Rank-Deficient Test Matrices with same K^{ii} (dwt_307) Sub-Matrices and RHS as Linear Combinations of Columns $C_1 + C_2 + C_3 + \cdots + C_n$ using Domain Decomposition

Sl. No	nsu	Size	Rank	ODU – DD $\sum\lvert x_i\rvert$ Error Norm	Original system ginverse MV $\sum\lvert x_i\rvert$ Error Norm	Original system geninv $\sum\lvert x_i\rvert$ Error Norm
1	5	2112x2112	1068	1.05579 1164 9.7350×10^{-13}	9.72518 1164 9.69202×10^{-12}	13.58581 1164 3.7717×10^{-11}
2	6	2464x2464	1246	1.21400 1358 1.0599×10^{-12}	15.7301 1358 1.0988×10^{-11}	21.73368 1358 5.7415×10^{-11}
3	7	2816x2816	1424	1.42343 1552 1.3897×10^{-12}	23.75027 1552 1.3625×10^{-11}	32.9830 1552 7.8649×10^{-11}

Table 5.30: Computational Times (in seconds) for Rank-Deficient Test Matrices with same K^{ii} (GD00_a) Sub-Matrices and RHS as Linear Combinations of Columns $C_1 + C_2 + C_3 + \cdots + C_n$ using Domain Decomposition

5.3.5 Problem Formulation with Different Sub-Matrices and RHS as Linear Combinations of Columns

The known right hand side vector is chosen as $C_1 + C_2 + C_3 + \cdots + C_n$ where n is the number of columns and C_i (with $i = 1,2,\dots,n$) represents the i^{th} column of the coefficient matrix $[G]$ in the big matrix.

Numerical results for the test cases are presented in Tables 5.31-5.32

Sl. No	nsu	Size	Rank	ODU – DD $\sum\lvert x_i\rvert$ Error Norm	Original system ginverse MV $\sum\lvert x_i\rvert$ Error Norm	Original system geninv $\sum\lvert x_i\rvert$ Error Norm
1	3	1228x1228	1152	1.04227 1228 3.9554×10^{-7}	4.381476 1228 2.7835×10^{-8}	6.44660 1228 2.4688×10^{-5}
2	4	1535x1535	1439	1.39427 1535 4.2179×10^{-7}	8.91303 1535 5.0078×10^{-8}	12.98718 1535 3.237×10^{-5}
3	5	1842x1842	1725	1.64756 1842 4.42625×10^{-7}	15.51327 1842 5.60808×10^{-8}	22.31969 1842 4.6407×10^{-5}

Table 5.31: Computational Times (in seconds) for Rank-Deficient Test Matrices with different K^{ii} (dwt_307) Sub-Matrices and RHS as Linear Combinations of Columns $C_1 + C_2 + C_3 + \cdots + C_n$ using Domain Decomposition

Sl. No	*nsu*	*Size*	*Rank*	*ODU – DD* $\sum\lvert x_i\rvert$ *Error Norm*	*Original system ginverse MV* $\sum\lvert x_i\rvert$ *Error Norm*	*Original system geninv* $\sum\lvert x_i\rvert$ *Error Norm*
1	15	1792x1792	1590	0.47628 1791.76 6.7768×10^{-12}	13.16311 1791.76 2.0569×10^{-10}	19.00465 1791.76 1.4315×10^{-9}
2	16	1904x1904	1686	0.52987 1903.66 7.9443×10^{-12}	15.83865 1903.66 2.9199×10^{-10}	22.6178 1903.66 7.944×10^{-12}
3	17	2016x2016	1781	0.57382 2015.52 7.2594×10^{-12}	18.88350 2015.52 3.3548×10^{-10}	26.91830 2015.52 2.16341×10^{-9}

Table 5.32: Computational Times (in seconds) for Rank-Deficient Test Matrices with different K^{ii} (GD98_c) Sub-Matrices and RHS as Linear Combinations of Columns $C_1 + C_2 + C_3 + \cdots + C_n$ using Domain Decomposition

6. EDUCATIONAL GENERALIZED INVERSE SOFTWARE FOR INTERNET USERS

Most of the currently available commercialized software (and/or freely available public source codes) for "large-scale" Engineering/Science computation has been written in either FORTRAN, C, or C++ languages. Legacy (commercialized) Finite Element Analysis (FEA) software, such as MSC-NASTRAN, SAP-2000, etc. have been written in FORTRAN language. While these languages are efficient for "number crunching", these FORTRAN/C/C++ software are NOT suitable for internet (educational) users. On the other hand, it is too time consuming if one has to re-write these (large) source codes in JAVA, or FLASH, etc... for internet/educational purposes. Based on our earlier research works [23], a general procedure for executing "any" FORTRAN software on the internet is explained and summarized in this chapter. More specifically, this chapter will explain how to use the developed FORTRAN generalized software.

6.1 Description for Executing FORTRAN Software on the Internet

Since the 1960's, scientific programs have been developed in FORTRAN for the solution of various structural, environmental, mathematical, chemical, etc. problems.

With the growing popularity and possibilities of the internet, web-based learning is becoming more and more popular these days. The new trend focuses on developing more effective learning methods for large pre-existing scientific languages like FORTRAN, C etc. In this chapter, a web-based environment is utilized as a means to introduce numerical methods concepts in civil engineering and other related fields of engineering. Software development and implementation is presented, including detailed descriptions of the techniques employed to link software written in high level computer languages, such as C and FORTRAN, to a web-based, user friendly interface for both input and output.

Web-based instruction systems represent a developing branch of computer-aided instruction. This type of educational information emphasizes the use of the web for transfer of educational information, and may be considered as a replacement to traditional delivery methods of lectures and textbooks.

The motivation for developing this educational software is the challenges faced while developing a 3-D truss analysis module in FLASH Actionscript language [24]. Initially, a web-based module was developed by converting 3-D truss analysis FORTRAN code to FLASH Actionscript language. This module analyzes a 3-D truss with the user specified input data. This module needs to re-write the entire FORTRAN code to a different language (in this case FLASH Actionscript). This process is not only time consuming; it requires a good knowledge of the other programming language (FLASH Actionscript). Fig. 6.1 shows a sample of the developed web based 3D truss module and can be found at http://www.lions.odu.edu/~skadi002/3d/. This emerged as one of the many situations where developed FORTRAN computer programs are no longer available for the public use (easy use) and are still considered to be valuable source of both research and educational material. A better and convenient way to interact the FORTRAN programs directly from the server is in needed to serve the purpose.

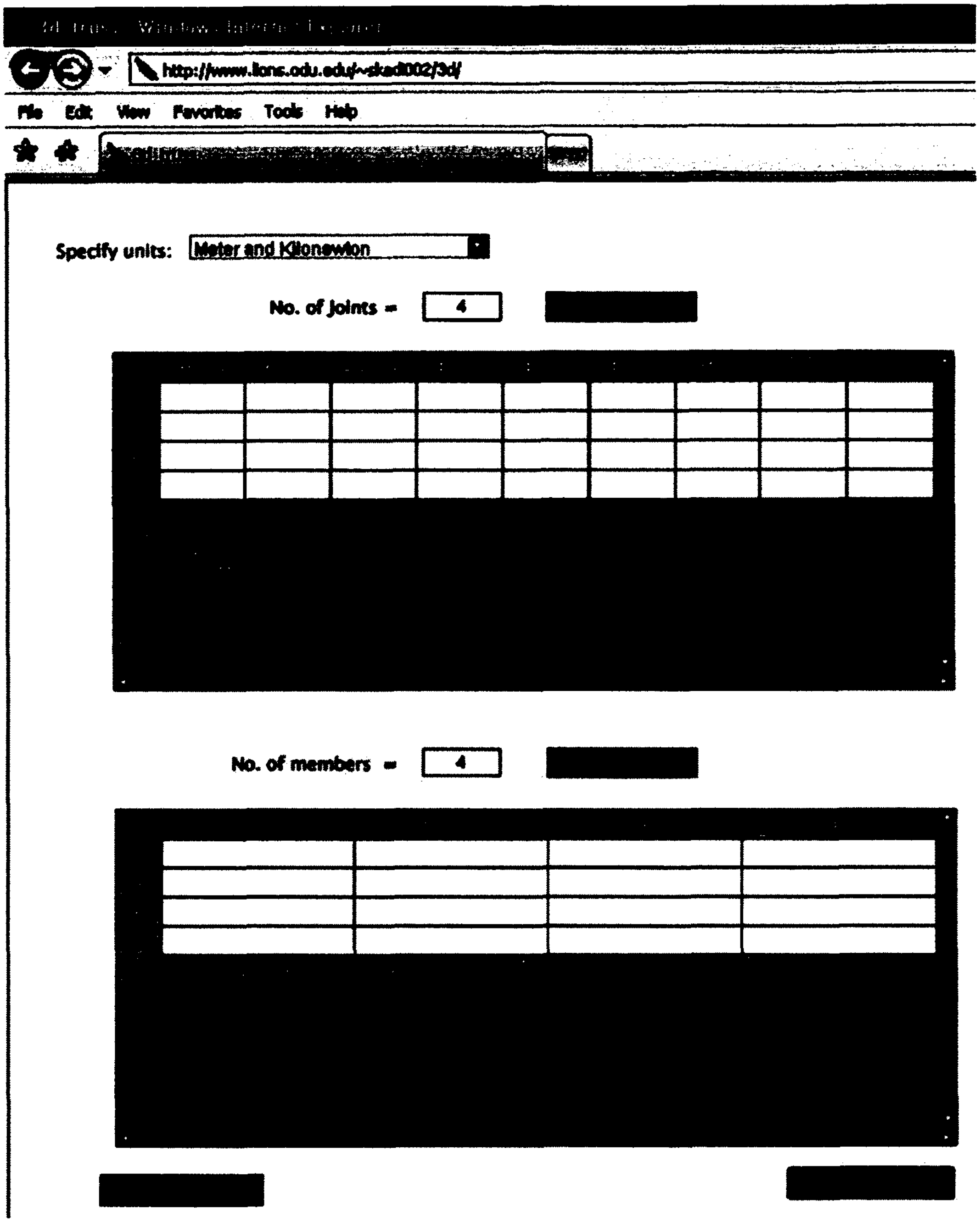

Figure 6.1: Online 3-D Truss Analysis.

6.2 Client Server Interface

Web application software is an application that uses a web browser as a client. Commonly used web browsers are Internet explorer, Mozilla Firefox, Google Chrome, Apple Safari, etc. A client is a system that accesses a remote service on another computer commonly known as a server.

Web applications commonly use a combination of server-side scripting languages like PHP, ASP, etc., and client-side languages like HTML, PHP, JAVA, etc. In the developed software, PHP [25] is used as both client side and server side technologies.

In Fig. 6.2 client can be any one of desktop, laptop, Mac PC or PDA with browser capabilities and an internet connection. The UNIX server is a machine where the compiled FORTRAN programs are stored.

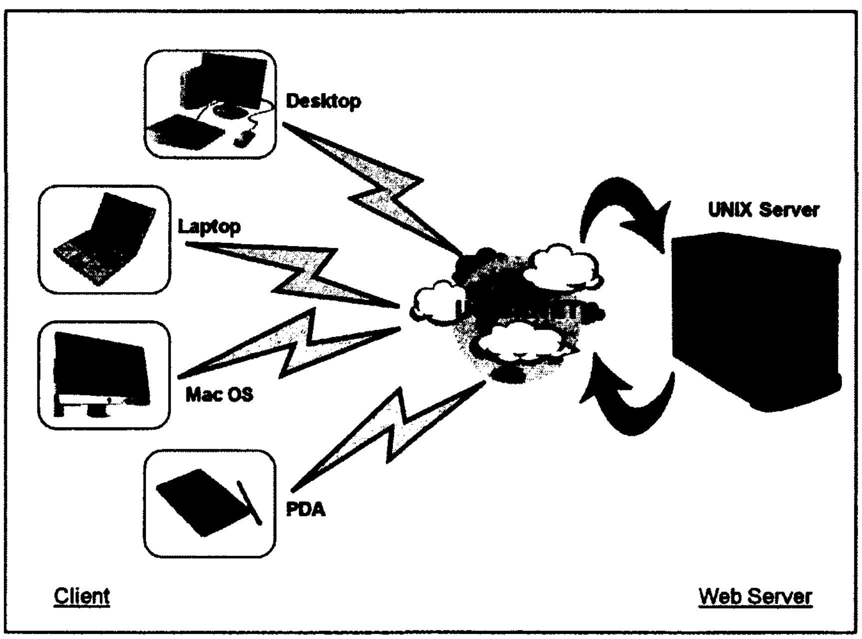

Figure 6.2: Client-Server Interface

6.3 Detailed Step-by-Step Procedures

Input and output interfaces are made user-friendly to attract users' attention, and guide them into use of the software in an efficient manner.

(a) The user references http://www.lions.odu.edu/~skadi002/work/ as URL which directs to a webpage containing number of educational applications. The user has a choice to select from various FORTRAN applications.

(b) For the sake of discussion, let us assume the user has interest in solving SLE (singular coefficient matrix) using generalized inverse. After clicking the appropriate link, the browser directs to a new webpage containing the description and usage of the application (see Fig. 6.3).

(c) The user clicks the "New" button to enter the input data in the textbox provided. It is a hypertext preprocessor (PHP) form that is interpreted by the browser to allow the user to enter the input data (see Fig 6.4).

(d) A sample/demo input data is provided in the input page so that the user can prepare in the specified fashion. Well documented instructions are provided for the users to prepare the input data to the application.

(e) Once the input data has been entered in the text box, the user hits the "Click here to submit input file" button which is located below the input textbox.

(f) At this point the input data has been sent to the FORTRAN program. The user clicks the "EXECUTE" button to execute/run the application.

(g) The output of the application is instantly seen on the same webpage (see Fig. 6.5)

6.4 Demonstrated Examples

In this section, various examples including the one explained in section 6.3 have been demonstrated. Below are the descriptions of the applications.

6.4.1 Solving system of Simultaneous Linear Equations using Generalized Inverse Algorithms

This application solves SLE of singular coefficient matrix using Generalized inverse algorithms. Fig. 6.3 shows a screenshot of the application homepage. The input data page with the sample of how input data page is prepared/entered is shown in Fig. 6.4. The output of the application is shown in Fig. 6.5.

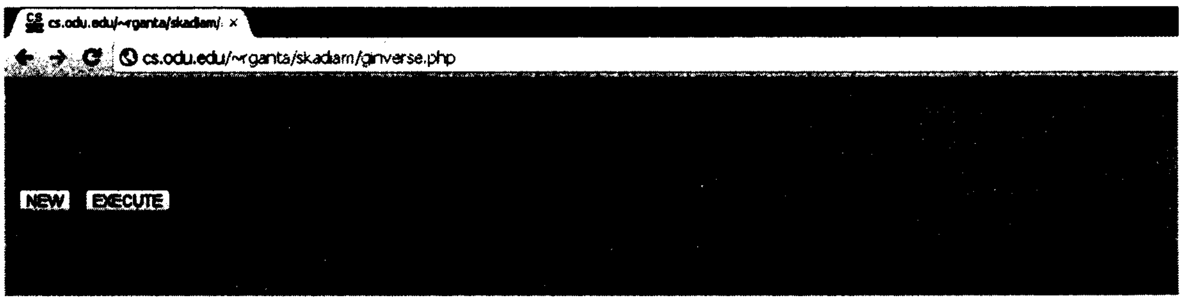

Figure 6.3 Sample of Generalized Inverse Home Page

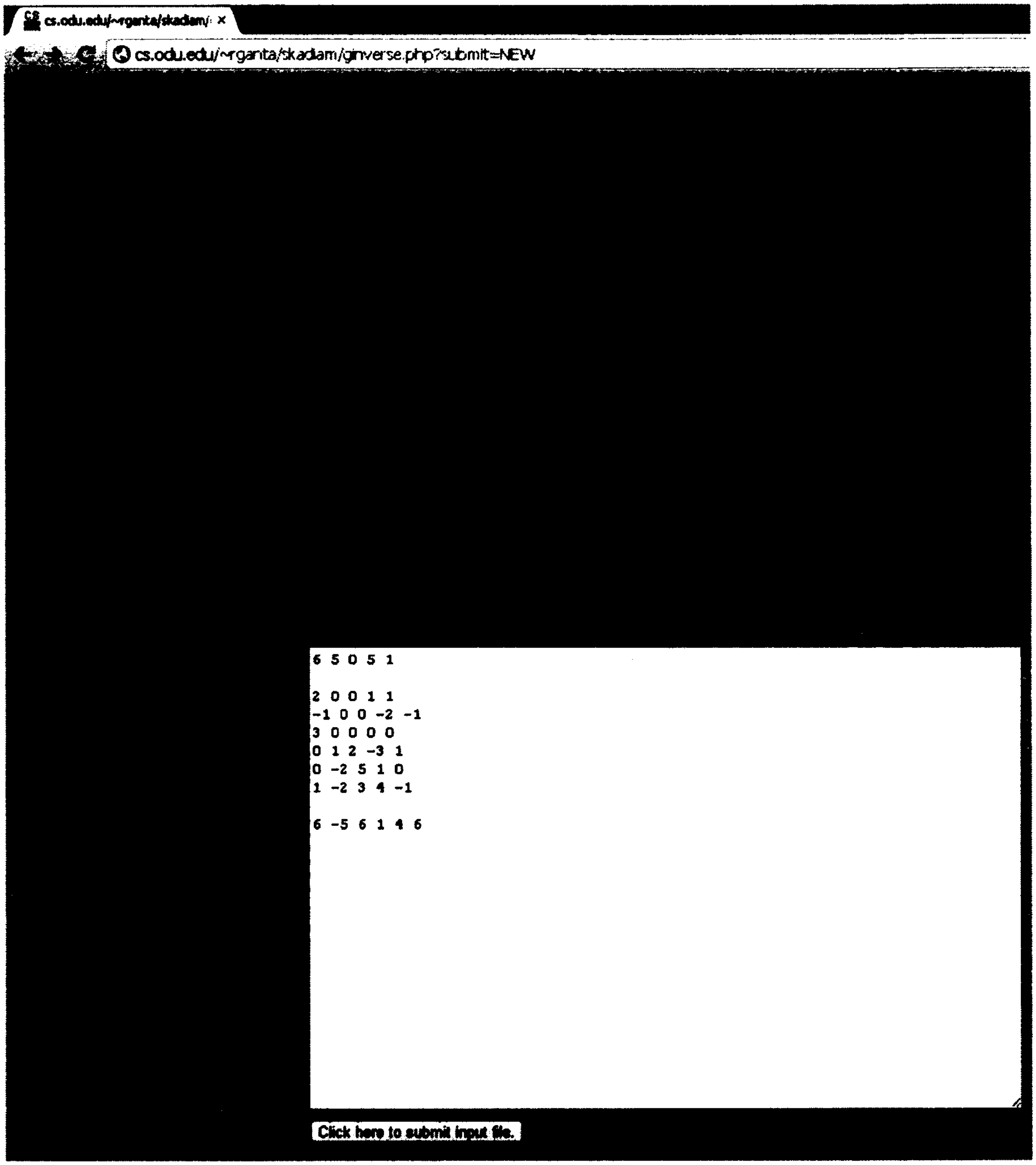

Figure 6.4 Generalized Inverse Input Page

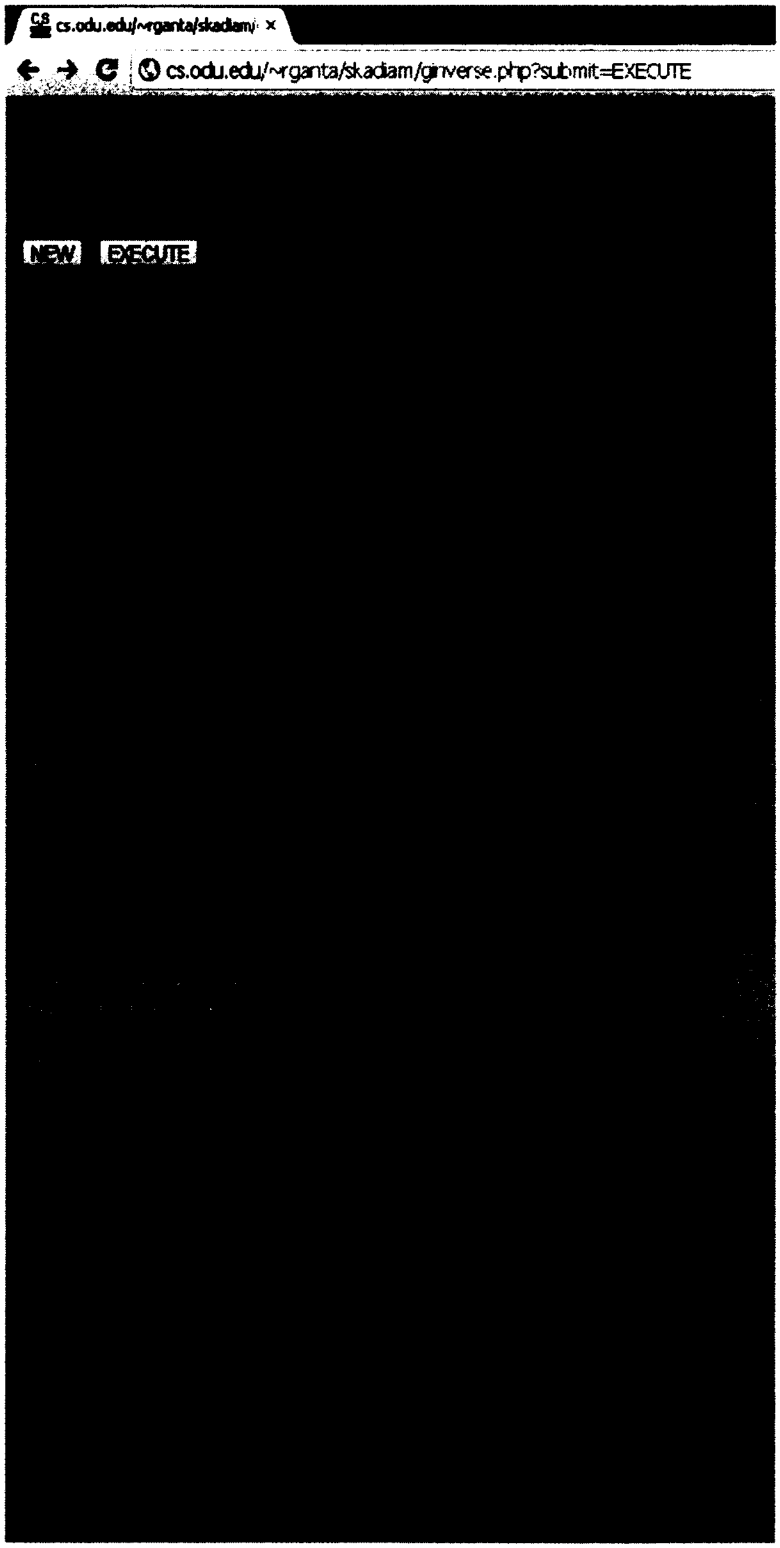

Figure 6.5 Generalized Inverse Output Page

6.4.2 Solving System of Simultaneous Linear Equations using LU Decomposition Algorithms

This application solves SLE of non-singular coefficient matrix using LU decomposition algorithms. Fig. 6.6 shows a screenshot of the application homepage. The input data page with the sample of how input data page is prepared is shown in Fig. 6.7. The output of the application is shown in Fig. 6.8.

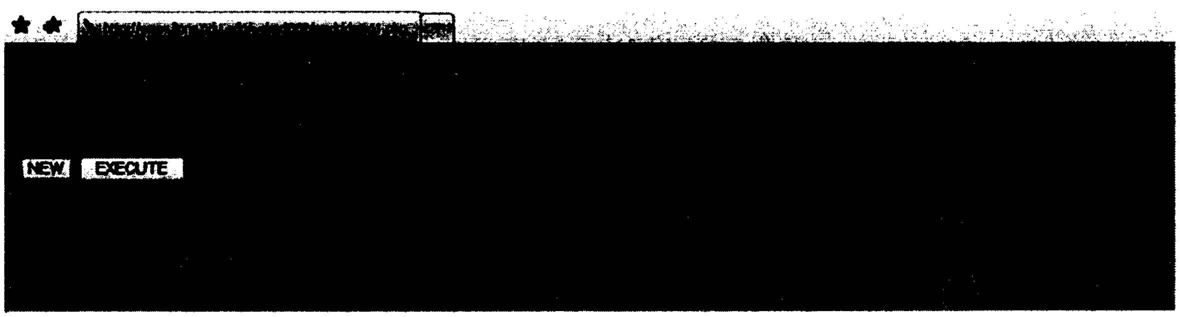

Figure 6.6: Sample of LU Decomposition Home Page.

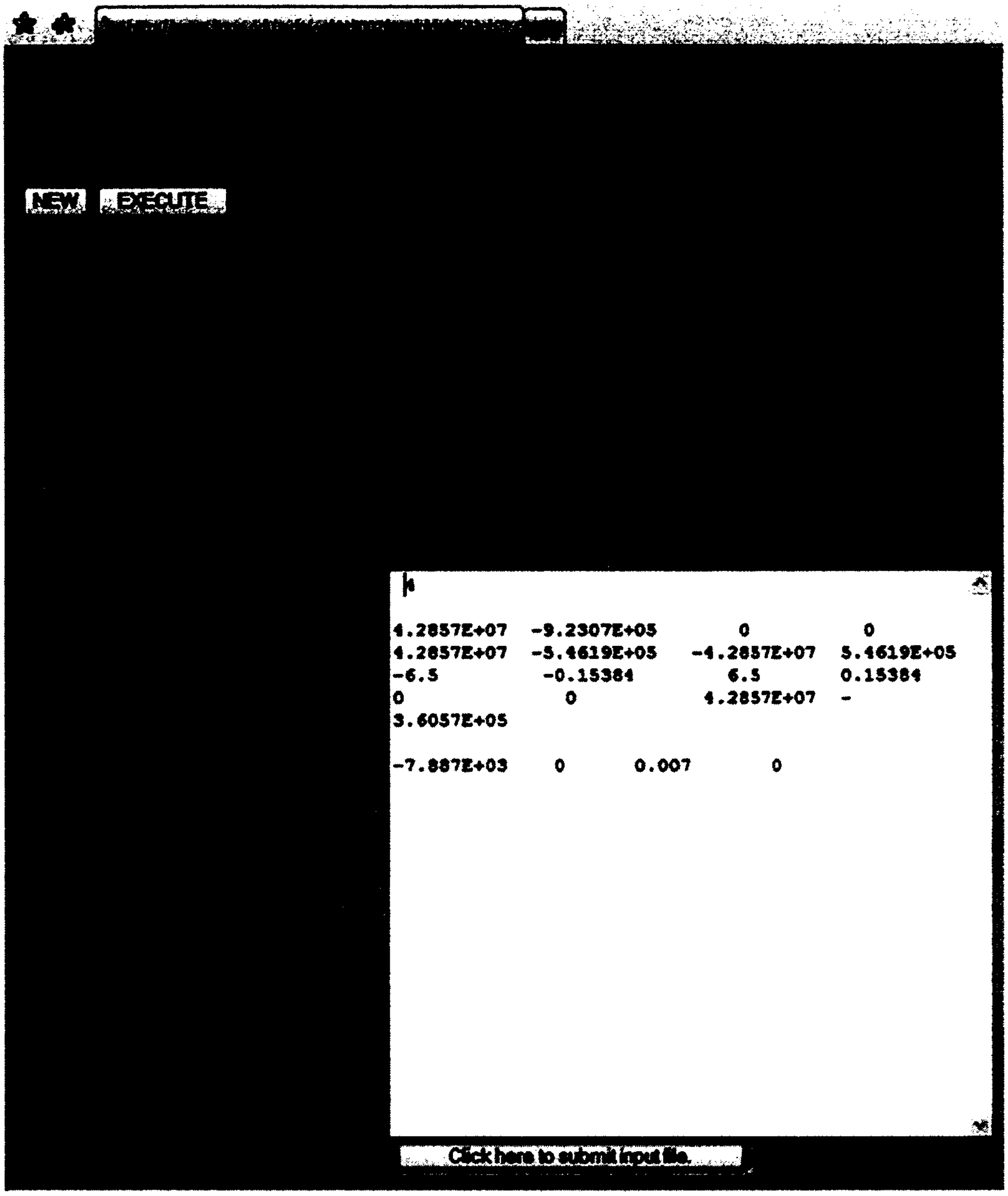

Figure 6.7: LU Decomposition Input Page

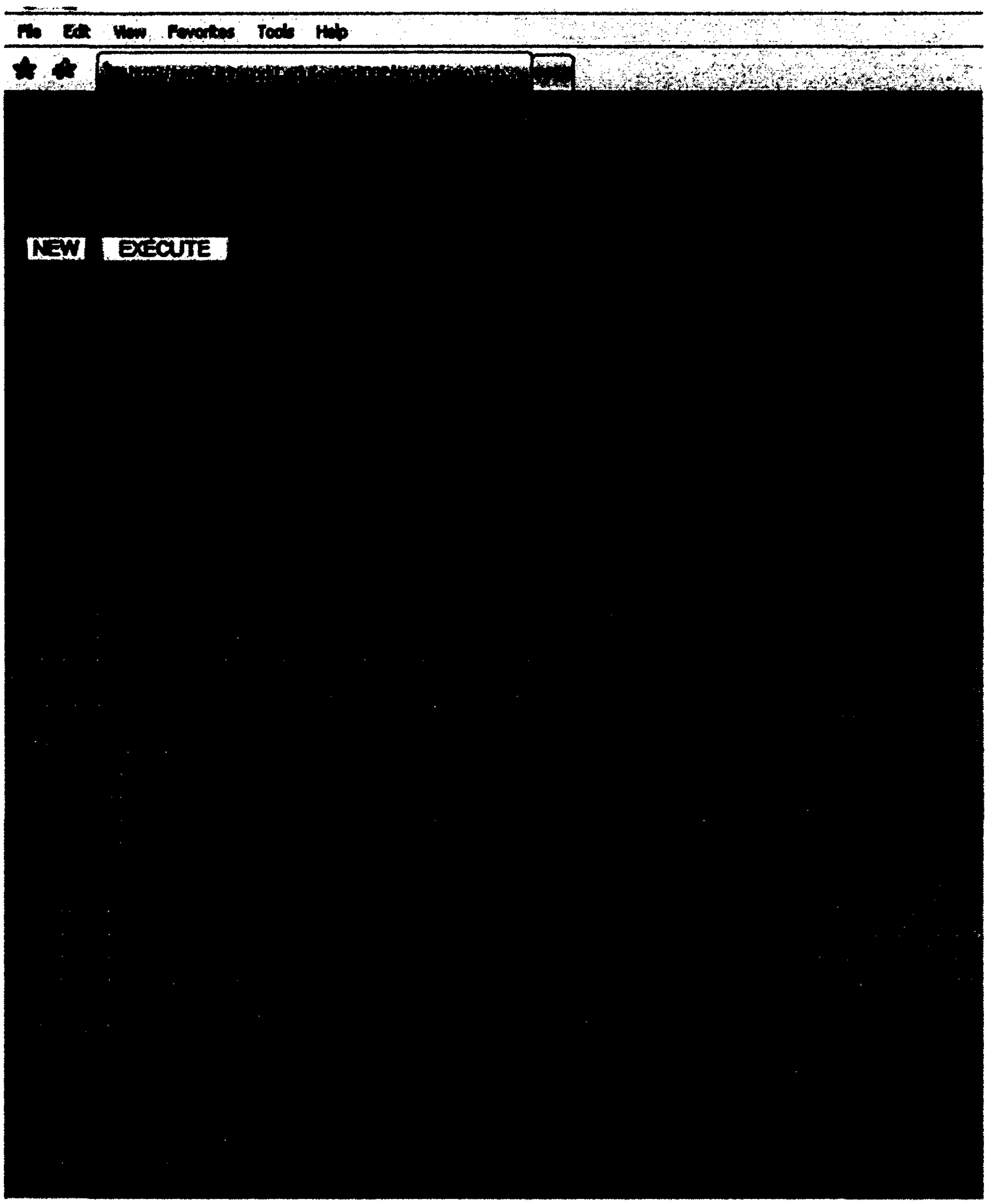

Figure 6.8: LU Decomposition Output Page

6.4.3 Solving System of Simultaneous Linear Equations using LU Decompositions Algorithms on Portable Devices

Nowadays, most of the portable devices such as smart phones, tablet PCs, personal digital assistants (PDA), e-book readers, etc., have internet browsing capabilities. The developed application can be accessed on the above said portable devices. One such example is shown in Fig. 6.9. In this example, the developed educational software is accessed on I-Phone.

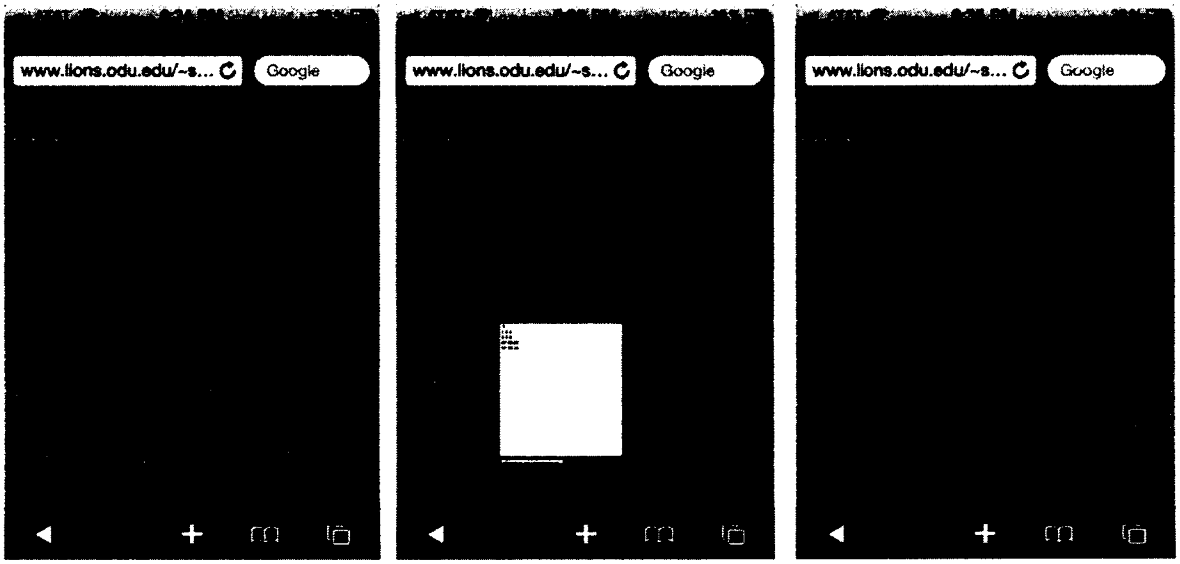

Figure 6.9: I-Phone Homepage, Input Data and Output Data.

7. MATLAB – MPI BUILT-IN FUNCTIONS FOR PARALLEL COMPUTING APPLICATIONS

7.1 Introduction

Matlab (MATrix LABoratory) is a tool to do numerical computations, solve engineering and sciences applications, display 2D and 3D graphical information, algorithm development, simulation, etc. It is a high-level scientific and engineering programming environment which provides many useful capabilities and has an extensive library of built-in functions.

Message Passing Interface (MPI) is widely used in large scale (intensive numerical) computations. This is especially true for "generalized inverse" computer implementation, where as matrix times matrix and/or Cholesky factorization operations were required. MPI is a library of functions/routines that can be used to create parallel programs in scientific languages such as FORTRAN, C, C++, etc.

Similar to traditional Message Passing Interface (MPI), MatlabMPI developed in Lincoln Laboratory, MIT, allows any Matlab programs to run in parallel. MatlabMPI implements the widely used MPI "look and feel" on the top of standard Matlab file input/output, resulting in Matlab implementation of MPI.

MatlabMPI can be downloaded into user's local ZORKA account from http://www.ll.mit.edu/mission/isr/matlabmpi/matlabmpi.html

7.2 MatlabMPI Functions

Some basic, most often used MatlabMPI built-in functions are briefly discussed below:

a. function MPI_Init

This function is called at the start of an MPI program. It also initializes MPI in Matlab environment.

Example:

MPI_Init;

b. function NP = MPI_Comm_size(comm)

This function returns the numbers of processors in the communicator. The "comm" in this function is an MPI communicator which is typically a copy of MPI_COMM_WORLD.

Example:

NP = MPI_Comm_size(comm)

c. function my_ID = MPI_Comm_rank(comm)

This function returns the rank (or processor ID #) of the current processor.

Example:

my_ID = MPI_Comm_rank(comm)

d. function MPI_Send(dest,tag,comm,varargin)

This function sends variable to a destination. It sends message containing variables to a specified destination with a given tag. The argument "dest" contains processor ID #, "tag" can be an any integer and "varargin" represents variable argument inputs.

Example:

MPI_Send (dest,tag,comm,data1,data2,data3,..)

e. function varargout = MPI_Recv(source,tag,comm)

This function receives a message from a specified source processor with a given tag and returns the output variable(s).

Example:

[var1,var2,var3,...] = MPI_Recv(source,tag,comm)

f. function MPI_Abort()

This function will abort any currently running MatlabMPI sessions by looking for leftover Matlab jobs and killing them.

g. function MPI_Finalize()

This function is the last statement indicating the end of a MatlabMPI program.

h. function MPI_Run(m_file,n_proc,machines)

This function runs a Matlab file by name "m_file" on multiple processors. It also runs "n_proc" number of copies of m_file on machines. To run on multiple processors, the argument "machines" are to be designated with "machine1, machine2,...'.

Example:

MPI_Run('example1',2,{}); for the case a single node (with 2 processors) is used.

MPI_Run('example2',4,{zorka1, zorka2}); for the case multiple nodes (assuming zorka1 and zorka2 are both available) are used.

The above discussed functions are used within traditional Matlab source code to run in parallel environment. In addition to the MPI functions, Matlab uses other built-in functions to perform various operations/tasks. Some of the additional functions are discussed below:

i. function eval()

This function execute string with Matlab expression and is also used with MPI_Run.

Example:

eval(MPI_Run('example1',2,{}))

j. function disp()

This function displays an array without printing the array name. It can also be used to display a string or a text inside the Matlab code.

Example:

disp(['Example1 from rank: ',num2str(my_rank)]);

If the rank is 0 (master processor), the output appears on the screen. And if the rank is more than 0, i.e 1,2,3... , the output prints to the corresponding file.

k. function MPI_Bcast(source,tag,comm,varargin)

This function broadcasts the variable(s) to all processors.

Example:

[var1, var2, ..] = MPI_Bcast(source,tag,comm,data1,data2,..)

7.3 Example 1: Display Rank of Processors

In this example, a simple MatlabMPI source code to print/display rank of different processors is shown below.

```
% MPI INITIALIZE

MPI_Init;

% MPI COMMUNICATOR

comm = MPI_COMM_WORLD;
```

% GET SIZE OR NUMBER OF PROCESSORS IN THE COMMUNICATOR

NP = MPI_Comm_size(comm)

% GET RANK (or ID #) OF CURRENT PROCESSOR

my_rank = MPI_Comm_rank(comm)

% DISPLAY RANK OF EACH PROCESSOR

disp(['Hello Message from rank: ', num2str(my_rank)]);

% FINALIZE Matlab MPI

MPI_Finalize;

% DISPLAY SUCCESS MESSAGE

disp('Success');

Let the file name for this MatlabMPI application be example1.m. In order to run in parallel environment, type the following statements in Matlab command prompt.

% ADDING PATH TO THE MatlabMPI SOURCE DIRECTORY TO INVOKE MPI FUNCTIONS.

addpath /local/MatlabMPI/src

% example1.m IS A MatlabMPI APPLICATION CODE WHICH NEEDS TO EXIST IN THE SAME WORKING DIRECTORY AS THE OTHER MPI FUNCTIONS. IN THIS CASE WE ARE USING 4 PROCESSORS.

eval(MPI_Run('example1',4,{}));

% ONCE THE DESIRED OUTPUT IS OBTAINED/PRINTED, THE FUNCTION MatMPI_Delete_all HAS TO BE INVOKED. THIS FUNCTION DELETES LEFTOVER MatlabMPI FILES FROM THE PREVIOUS RUN. THIS FUNCTION IS ALSO INVOKED BEFORE THE START OF A NEW MatlabMPI APPLICATION.

MatMPI_Delete_all;

7.4 Example 2: Matrix-Matrix Multiplication

The MatlabMPI source code for matrix-matrix multiplication (dense format) can be found in Appendix D.

Below are the time results for matrix times matrix multiplication (size = 1000).

Sl.No	Number of Processors (NP)	Time (seconds)
1	2	39.4087
2	4	15.7884
3	6	11.7958
4	8	9.5475
5	10	8.3805
6	12	7.5807

Table 7.1 Time Results (in seconds) for Matrix-Matrix Multiplication using MatlabMPI

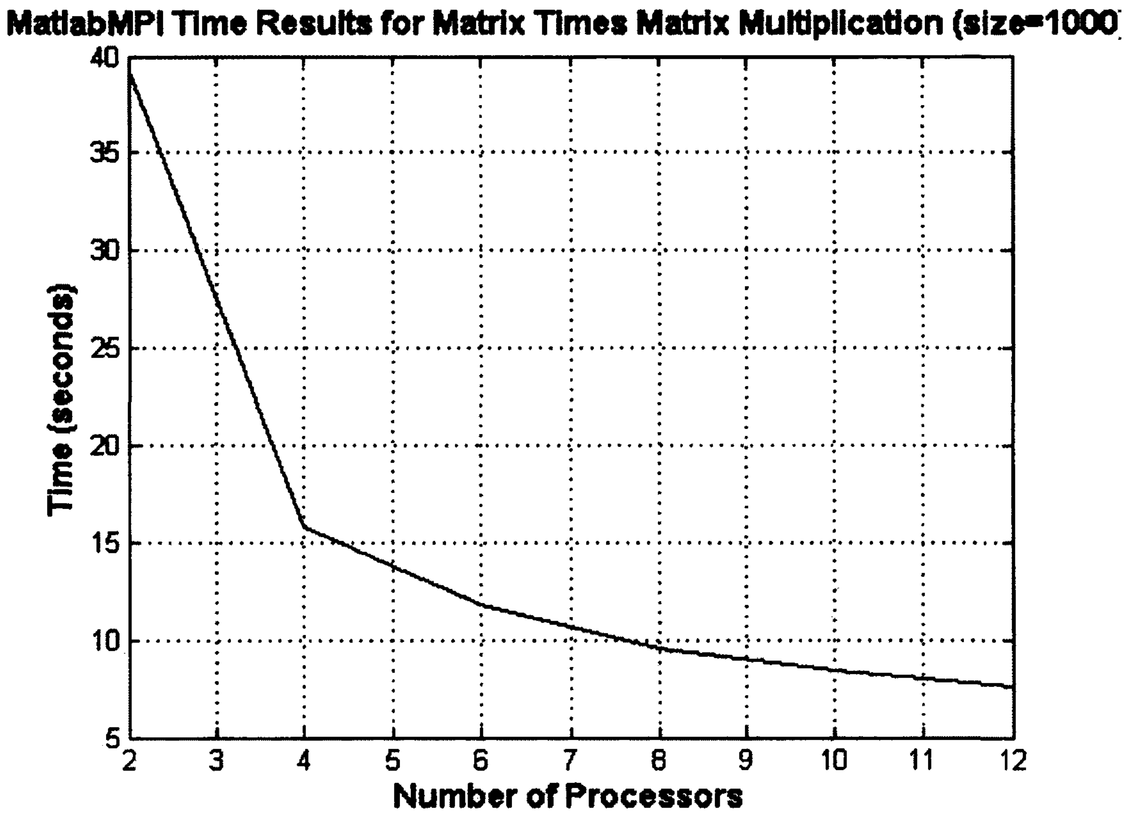

Figure 7.1 Graphical Representation of Time Results (in seconds) for Matrix-Matrix Multiplication using MatlabMPI

8. CONCLUSIONS AND FUTURE WORKS

In this dissertation, various efficient algorithms for solving SLE with full rank or rank deficient have been reviewed, proposed and tested. These algorithms were based on efficient generalized inverse algorithms, which had also been incorporated into the DD formulation. Users are provided the options of incorporating either direct, or iterative solvers into the developed DD generalized inverse formulation. Extensive numerical results have been used to evaluate the performance (in terms of numerical accuracy, calculated error norm, CPU/wall-clock time) of the proposed procedures. The developed numerical procedures can be applied to solve "general" SLE (in the form $[G]\{x\} = \{b\}$, where the coefficient matrix $[G]$ could be square/rectangular, symmetrical/unsymmetrical, non-singular/singular). Numerical results have shown that the proposed algorithms are highly efficient as compared to existing algorithms [6, 9, 13] (including the popular MATLAB built-in function $pinv(G) * b$) [12]. Further reduction in wall-time can also be realized/achieved by taking advantages of "parallel matrix times matrix operations" under MATLAB-MPI computer environment [26].

Furthermore, this dissertation has also contributed the "educational value" to the educational communities, by providing the tools/technologies to execute any existing FORTRAN code for internet users, without requiring them to download any (commercial) software on their desktop/laptop computers. The only requirement for the users to use/learn/execute our FORTRAN-web application is to have access to the internet, which is readily available not only in every home, but also in most public places (such as in the airports, hotels, universities, restaurants, etc.).

Extensions to this current work may include a variation of DD formulation proposed in [4], parallel implementation of the proposed DD generalized inverse solver, and incorporating METiS [20] reordering algorithm for automatically partitioned a given coefficient matrix into diagonal blocks, in such a way to minimize the total number of boundary (interface) nodes, etc.

REFERENCES

1. Greenbaum, A. (1997). *Iterative Methods for Solving Linear Systems, Frontiers in Applied Mathematics*, SIAM, Philadelphia.
2. Farhat, C., Roux, F.X. and Oden, J.T. (1994). *Implicit Parallel Processing in Structural Mechanics*. International Association for Computational Mechanics Advances. Vol. 2., North-Holland publisher.
3. Heath, M.T. (1997). *Scientific Computing: An Introductory Survey*. McGraw-Hill, New York.
4. Hou, G. and Y. Wang, *A Substructuring Technique For Design Modifications of Interface Conditions*, Old Dominion University, Norfolk. (Personal Conversation)
5. Rakha, M.A. (2004). "On the Moore-Penrose Generalized Inverse Matrix." *Applied Mathematics and Computation,* 158, 185-200.
6. Katsikis, V.N. and Pappas, D. (2008). "Fast Computing of the Moore-Penrose Inverse Matrix." *Electronic Journal of Linear Algebra*, 17, 637-650.
7. Kucera, R., Kozubek, T., Markopoulos, A., and Machalova, J. (2012). "On the Moore-Penrose Inverse in Solving Saddle-Point Systems with Singular Diagonal Blocks." *Numerical Linear Algebra with Applications*, 19, 677-699.
8. Brzobohaty, T., Dostal, Z., Kozubek, T., Kovar, P., and Markopoulos, A. (2011). "Cholesky Decomposition with Fixing Nodes to Stable Computation of a Generalized Inverse of the Stiffness Matrix of a Floating Structure." *International Journal For Numerical Methods in Engineering,* 88, 493-509.

9. Chen, X. and Ji, Ji. (2011). "Computing the Moore-Penrose Inverse of a Matrix through Symmetric Rank-One Updates." *American Journal of Computational Mathematics,* 1, 147-151.

10. Golub, G.H. and Loan, C.F.V. (1996). *Matrix Computations*, The John Hopkins University Press.

11. Nguyen, D.T., *Finite Element Methods: Parallel-Sparse Statics and Eigen-Solutions*, Springer.

12. MATLAB, *MATLAB – The Language of Technical Computing.*

13. Pierre, C. (2005). "Fast Computation of Moore-Penrose Inverse Matrices." *Neural Information Processing – Letters and Reviews*, 8(2).

14. Davis, T. *University of South Florida Matrix Collection.*

15. SJSU. *SJSU Singular Matrix Database.*

16. Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems.* 2nd ed, SIAM, Minnesota

17. Israel, B. and Greville, T.N.E. (1980). *Generalized Inverses Theory and Applications.* 2nd ed, Krieger.

18. Rao, C.R, and Mitra, S.K. (1972). *Generalized Inverse Matrices and its Applications.* Wiley Series in Probability and Mathematical Statistics.

19. Duc T Nguyen, Kaw, A. and Kalu, E. (2011). *Numerical Methods with Applications customized for Old Dominion University.*

20. Metis. *http://glaros.dtc.umn.edu/gkhome/views/metis.*

21. Pissanetsky, S. (1984). *Sparse Matrix Technology.* Academic Press

22. George, A., Liu, J.W.H., and Gilbert, J.R. (1993) *Graph Theory and Sparse Matrix Computation*. Springer.

23. Kadiam, S., Mohammed, A. and Nguyen, D.T. (2011) "Development of Web-Based Engineering, Educational and Assessment Modules for Learning Numerical Methods." Proc., *VMASC Student Capstone*. Va.

24. FLASH ACTIONSCRIPT, *http://www.adobe.com/devnet/actionscript.html.*

25. PHP. *http://php.net/manual/en/index.php.*

26. Kepner, D.J. *Parallel Programming with MatlabMPI. http://www.ll.mit.edu/mission/isr/matlabmpi/matlabmpi.html.*

APPENDIX A

SINGULAR VALUE DECOMPOSITION (SVD) AND THE GENERALIZED INVERSE

Example A1

Given $A = \begin{bmatrix} 2 & 3 \\ 4 & 6 \end{bmatrix}$ (A.1)

Step1: Compute $AA^H = AA^T = \begin{bmatrix} 2 & 3 \\ 4 & 6 \end{bmatrix}\begin{bmatrix} 2 & 4 \\ 3 & 6 \end{bmatrix} = \begin{bmatrix} 13 & 26 \\ 26 & 52 \end{bmatrix}$ (A.2)

Also: $AA^H = (U\Sigma V^H)(V\Sigma^H U^H) = U\Sigma^2 U^H$

Similarly, $A^H A = V\Sigma^2 V^H$

Also compute $A^H A = A^T A = \begin{bmatrix} 2 & 4 \\ 3 & 6 \end{bmatrix}\begin{bmatrix} 2 & 3 \\ 4 & 6 \end{bmatrix} = \begin{bmatrix} 20 & 30 \\ 30 & 45 \end{bmatrix}$ (A.3)

Step2: Compute the standard Eigen-solution individually for $AA^T and\ A^T A$

Using MATLAB built-in function "eig ", Eigen-values and the corresponding Eigen-vectors for AA^T are given as

$[u1, lamba1] = eig(AA^T)$

$u1 = \begin{bmatrix} -0.8944 & 0.4472 \\ 0.4472 & 0.8944 \end{bmatrix} \equiv U$ (A.4)

$lambda1 = \begin{bmatrix} 0 & 0 \\ 0 & 65 \end{bmatrix}$ (A.5)

Also, for $A^T A$

$[u2, lamba2] = eig(A^T A)$

$u2 = \begin{bmatrix} -0.8321 & 0.5547 \\ 0.5547 & 0.8321 \end{bmatrix} \equiv V$ (A.6)

$lambda2 = \begin{bmatrix} 0 & 0 \\ 0 & 65 \end{bmatrix}$ (A.7)

From the above equations, we can observe that Eigen-values in both the cases are the same, but their corresponding Eigen-vectors are different.

Also, $\sigma = \sqrt{lambda1 = lambda2}$

Computing $\sigma = \sqrt{lambda1 = lambda2} = \begin{bmatrix} 0 & 0 \\ 0 & \sqrt{65} \end{bmatrix}$ (A.8)

From Eqs. (A.4, A.7 and A.8), the SVD of Eq. (A.1) can be obtained as

$$A = U\Sigma V = \begin{bmatrix} -0.8944 & 0.4472 \\ 0.4472 & 0.8944 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & \sqrt{65} \end{bmatrix} \begin{bmatrix} -0.8321 & 0.5547 \\ 0.5547 & 0.8321 \end{bmatrix} \quad (A.9)$$

The generalized inverse A^{+} of Eq. (A.1) is computed as

$$A^{+} = V\Sigma^{+}U^{H} \quad (A.10)$$

where $\Sigma^{+} = \begin{bmatrix} 0 & 0 \\ 0 & \frac{1}{\sqrt{65}} \end{bmatrix}$ (A.11)

Hence,

$$A^{+} = \begin{bmatrix} -0.8321 & 0.5547 \\ 0.5547 & 0.8321 \end{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & \frac{1}{\sqrt{65}} \end{bmatrix} \begin{bmatrix} -0.8944 & 0.4472 \\ 0.4472 & 0.8944 \end{bmatrix} \quad (A.12)$$

$$A^{+} = \begin{bmatrix} 0.0308 & 0.0615 \\ 0.0462 & 0.0923 \end{bmatrix} \quad (A.13)$$

The result obtained in Eq. (A.13) has been checked with MATLAB generalized inverse function $pinv()$ and same result is obtained.

APPENDIX B

AN EDUCATIONAL FORTRAN SOURCE CODE OF "SPECIAL LDL^T" ALGORITHM FOR FACTORIZATION OF SINGULAR/SQUARE/SYMMETRICAL COEFFICIENT MATRIX

```
c
      Implicit real*8 (a-h, o-z)
c
c=====================================================================
c
c     Remarks :
c     (a)    Identifying which are dependent rows of a "floating" substructure
c     (b)    Factorizing (by LDL_transpose) of a floating substructure stiffness
c            Whenever a dependent row is encountered during LDL factored
c            process, then we just :
c            [1] set all factorized values of the dependent row to be ZEROES
c            [2] ignore the dependent row(s) in all future faztorized rows
c     (c)    [K "float"] = [K11]    [K12]
c                          [K21]    [K22]
c            where [K11] = full rank ( =non-singular )
c     (d)    The LDL_transpose of [K11] can be obtained by taking the results
c            of part (b) and deleting the dependent rows/columns
c     Author(s) : Prof. Duc T. Nguyen
c     Version :  04-30-2004 (EDUCATIONAL purpose, LDL/FULL matrix is
assumed)
c     Stored at : cd ~/cee/*odu*clas*/generalized_inverse_by_ldl.f
c
c=====================================================================
c
      dimension u(99,99), idepenrows(99), tempo1(99)
c
      iexample=1              ! can be 1, or 2, or 3
c
      if (iexample . eq. 1)   n=3
c
if (iexample . eq. 2)   n=12
c
      if (iexample . eq. 3)   n=7
c
      do 1 i=1,n
      do 2 j=1,n
      u(i,j)=0
 2    continue
 1    continue
```

```
c
      if (iexample . eq. 1)    then
c
      u(1,1)= 2.               ! non-singular case
c     u(1,1)= 1.               !       singular case
      u(1,2)= -1.
      u(2,2)= 2.
      u(2,3)= -1.
      u(3,3)= 1.
c
      elseif (iexample . eq. 2)    then
c
      u(1,1)= 1.88*10**5
      u(1,2)= -4.91*10**4
      u(1,3)= -1.389*10**5
      u(1,7)= -4.91*10**4
      u(1,8)= 4.91*10**4
c
      u(2,2)= 1.88*10**5
      u(2,6)= -1.389*10**5
      u(2,7)= 4.91*10**4
      u(2,8)= -4.91*10**4
c
      u(3,3)= 1.88*10**5
      u(3,4)= 4.91*10**4
      u(3,5)= -4.91*10**4
      u(3,6)= -4.91*10**4
c
      u(4,4)= 1.88*10**5
      u(4,5)= -4.91*10**4
      u(4,6)= -4.91*10**4
      u(4,8)= -1.389*10**5
c
      u(5,5)= 2.371*10**5
      u(5,7)= -1.389*10**5
      u(5,11)= -4.91*10**4
      u(5,12)= 4.91*10**4
c
      u(6,6)= 3.76*10**5
      u(6,10)= -1.389*10**5
      u(6,11)= 4.91*10**4
      u(6,12)= -4.91*10**4
c
      u(7,7)= 2.371*10**5
      u(7,9)= -4.91*10**4
      u(7,10)= -4.91*10**4
```

```
c
      u(8,8)= 3.76*10**5
      u(8,9)= -4.91*10**4
      u(8,10)= -4.91*10**4
      u(8,12)= -1.389*10**5
c
      u(9,9)= 1.88*10**5
      u(9,10)= 4.91*10**4
      u(9,11)= -1.389*10**5
c
      u(10,10)= 1.88*10**5
c
      u(11,11)= 1.88*10**5
      u(11,12)= -4.91*10**4
c
      u(12,12)= 1.88*10**5
c
      elseif (iexample . eq. 3)    then
c
      u(1,1)= 1.
      u(1,2)= 2.
      u(1,3)= -3.
      u(1,4)= 2.
      u(1,5)= -2.
      u(1,6)= -3.
      u(1,7)= -2.
c
      u(2,2)= 4.
      u(2,3)= -6.
      u(2,4)= 4.
      u(2,5)= -4.
      u(2,6)= -6.
       u(2,7)= -4.
c
      u(3,3)= 9.
      u(3,4)= -6.
      u(3,5)= 6.
      u(3,6)= 9.
      u(3,7)= 6.
c
      u(4,4)= 5.
      u(4,5)= -1.
      u(4,6)= -5.
      u(4,7)= -7.
c
      u(5,5)= 13.
```

```
      u(5,6)= 9.
      u(5,7)= -5.
c
      u(6,6)= 13.
      u(6,7)= 9.

      u(7,7)= 27.
      Endif
c
      do 4 i=1,n
      do 5 j=1,n
      u(j,i)=u(i,j)
 5    continue
 4    continue
c
call generalized_inverse_ldl (n, u, idependrows, ndependrows)
c
      write(6,*) '# dependent rows = ' ,ndependrows
      if (ndependrows .ge. 1)    then
      write(6,*) ' dependent rows = ' ,(idependrows(i) ,i=1, ndependrows)
      endif
c     write(6,*) ' LDL factorized u(- , -) =', ((u(i,j) ,j=i,n) ,i=1,n)
c     extracting & writing the LDL factorized of full rank of [K11]
c     by deleting the dependent row(s) /column(s) of [u]
        do 52 i=1,n
        iskiprow=0
          do 53 j=1,ndependrows
          if (idependrows(j) .eq. i)    iskiprow=1
 53       continue
        if (iskiprow .eq. 1)    go to 52
          icount=0
          do 54 j=i,n
          iskipcol=0
            do 55 k=1,ndependrows
            if (idepenrows(k) .eq. 0)    iskipcol=1
 55         continue
          if (iskipcol .eq. 0)    then
          icount=icount+1
          tempo1(icount)=u(i,j)
          endif
 54       continue
            write(6,*) 'LDL of [K11] = ' ,(tempo1(k) ,k=1,icount)
 52     continue
c
stop
      end
```

```
c
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
c
      subroutine generalized_inverse_ldl (n, u, idependrows, ndependrows)
      Implicit real*8 (a-h, o-z)
      dimension u(99,*), idepenrows(*)
c
c=====================================================================
c
c     Remarks :
c     (a)    Identifying which are dependent rows of a "floating" substructure
c     (b)    Factorizing (by LDL_transpose) of a floating substructure stiffness
             Whenever a dependent row is encountered during LDL factored
c            process, then we just :
c            [1] set all factorized values of the dependent row to be ZEROES
c            [2] ignore the dependent row(s) in all future faztorized rows
c     (c)    [K "float"] = [K11]    [K12]
c                          [K21]    [K22]
c            where [K11] = full rank ( =non-singular )
c     (d)    The LDL_transpose of [K11] can be obtained by taking the results
             of part (b) and deleting the dependent rows/columns
c     Author(s) : Prof. Duc T. Nguyen
c     Version :   04-30-2004
c     Stored at : cd ~/cee/*odu*clas*/generalized_inverse_by_ldl.f
c
c=====================================================================
c
  eps=0.0000000001
      do 11 i=2,n
        do 22 k=1,i-1
        if (dabs( u(k,k) ) .lt. eps)    go to 22        ! check for "previous"
c                                                       ! dependent row(s)
        xmult=u(k,i) /u(k,k)
          do 33 j=i,n
          u(i,j)=u(i,j) -xmult*u(k,j)
 33       continue
        u(k,i)=xmult
 22     continue
c
c=====================================================================
c
c     to zero out entire dependent row
        if (dabs( u(i,i) ) .lt. eps)    then
        write(6,*) 'dependent row # i, u(i,i) = ' ,i ,u(i,i)
        ndependrows= ndependrows+1
        idependrows(ndependrows)= i
```

```
          do 42 j=i,n
 42       u(i,j)=0.
            do 44 k=1,i-1
44          u(k,i)=0.
        endif
c
c=====================================================================
c
 11     continue
c
return
        end
c
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

L D L_t factorized of the "full rank" sub-matrix [K11] of Example 3

```
dependent row # i, u(i,i) =   2   0.0E+0
dependent row # i, u(i,i) =   3   0.0E+0
dependent row # i, u(i,i) =   5   0.0E+0
# dependent rows =   3
dependent rows =   2   3   5

LDL of [K11]  =  1.0  2.0  -3.0  -2.0
LDL of [K11]  =  1.0  1.0  -3.0
LDL of [K11]  =  3.0  2.0
LDL of [K11]  =  2.0
```

++

L D L_t factorized of the "full rank" sub-matrix [K11] of Example 2

```
dependent row # i, u(i,i) =   10   -8.731149137020111E-11
dependent row # i, u(i,i) =   11   -5.820766091346741E-11
dependent row # i, u(i,i) =   12   -2.910383045673370E-11
# dependent rows =   3
dependent rows =   10   11   12
LDL of [K11]  =  188000.0  -0.26117002127659574  -0.7388297872340426  0.0E+0
0.0E+0  0.0E+0  -0.26117002127659574  0.26117002127659574  0.0E+0
LDL of [K11]  =  175176.54255319148  -0.2070856178827499  0.0E+0  0.0E+0
-0.7929143821172502  0.2070856178827499  -0.2070856178827499  0.0E+0
LDL of [K11]  =  77864.19232391396  0.6305851063829787  -0.6305851063829787
-1.0  -0.36941489361702123  0.36941489361702123  0.0E+0
LDL of [K11]  =  157038.27127659574  -0.11550223476828982  0.0E+0
0.11550223476828982  -1.0  0.0E+0
```

LDL of [K11] = 204043.26040931543 -0.24063524519998494 -
0.7593647548000151 0.0E+0 0.0E+0
LDL of [K11] = 176184.80946068066 -0.211623292466662 -
2.0648651936724454E-17 0.0E+0
LDL of [K11] = 78494.47532361932 0.6255217300016221 -0.6255217300016221
LDL of [K11] = 157286.88305692037 -0.11690029517761087
LDL of [K11] = 155137.45100017014

++

L D L_t factorized of the "full rank" sub-matrix [K11] of Example 1 (singular case)

dependent row # i, u(i,i) = 3 0.0E+0
dependent rows = 1
dependent rows = 3

LDL of [K11] = 1.0 -1.0
LDL of [K11] = 1.0

++

L D L_t factorized of the "full rank" sub-matrix [K11] of Example 1 (non-singular case)
dependent rows = 0

LDL of [K11] = 2.0 -0.5 0.0E+0
LDL of [K11] = 1.5 -0.6666666666666666
LDL of [K11] = 0.33333333333333337

++

APPENDIX C

A COMPLETE LISTING OF AN EDUCATIONAL FORTRAN SOURCE CODE OF "CHOLESKY GENERALIZED INVERSE" ALGORITHMS FOR SLE

```
      implicit real*8(a-h,o-z)
      real tar(2)
      integer r, ractual
c=====================================================================
c......Remarks:
c     (a) Identifying which are dependent rows of a "floating" substructure
c     (b) Factorizing (by cholesky) of floating substructure stiffness
c         Whenever a dependent row is encountered during cholesky factored process,
c         then we just:
c         [1] set all factorized values of the dependent row to be ZEROES
c         [2] ignore the dependent row(s) in all future factorized rows
c     (c) [K "float"] = [K11]  [K12]
c                       [K21]  [K22]
c         where [K11] = full rank ( = non-singular )
c                     = {K "float"] with deleting the dependent rows/columns
c     (d) The u_transpose * U  of [K11] can be obtained by taking the results
c         of part (b) and deleting the dependent rows/columns
c......Author(s):  Prof. Duc T. Nguyen
c......Version:  02-11-2012 (EDUCATIONAL purpose,LDL/FULL matrix is assumed)
c......Stored at:  cd ~/cee/*odu*clas*/generalized_inverse_by_cholesky.f
c
c......Notes: Prof. Duc Nguyen's "generalized cholesky" code has been correctly
c......      verified for AT LEAST 7-8 different examples
c......      [see generalize*cholesky*.dat;  and see out1-keep]
c=====================================================================
      dimension u(1999,1999),idependrows(1999),tempo1(1999),
     $          am_inv(1999,1999), tempo2(1999)
      dimension ut(1999,1999), am(1999,1999), amt(1999,1999),
     $          g(1999,1999), gt(1999,1999)
      dimension itempo1(1999), rhs(1999)
c
       call pierrotime(t1)
c
      write(6,*) '                                  '
      write(6,*) '================================================'
      write(6,*) 'today date: 04-23-2012; Prof. Duc T. Nguyen'
      write(6,*) '================================================'
      write(6,*) '                                  '
c
      maxnr1 = 1999
```

```
      maxnc1 = 1999
      maxnc2 = 1999
c
c......input (or randomly generate rectangular/square matrix [G] of dimension mxn
c......where we assume/prefer  m > n
c
      read(5,*) m, n, iautodata, irankn, iaxeqb
      write(6,*) 'user input: m,n,iautodata,irankn,iaxeqb = '
      write(6,*)  m,n,iautodata,irankn,iaxeqb
      write(6,*) '                          '
c
c......read user's input matrix data
c
      if (iautodata .eq. 0) then
       do 32 i=1, m
        read(5,*) (g(i,j), j=1,n)
 32     continue
c
c......user's input rhs vector {rhs} nx1
c
        read(5,*) (rhs(i), i=1,m)
c
c......randomly generated input matrix data
c
      elseif (iautodata .eq. 1) then
       ndependcols = n - irankn
       icount = 0
       idum=0
       do 61 j=1,n
       if (j .LE. irankn) then
       irandom_col = irand(1, n)
c       write(6,*) 'irandom_col = ', irandom_col
        do 60 i=1, m
        g(i,j) = drand(idum) * 10000.d0
 60     continue
       elseif (j .GT. irankn) then
        do 66 i=1, m
        g(i,j) = 0.d0
 66     continue
       endif
 61     continue
c
c......generated rhs vector {rhs} nx1, such that solution vector = {1, 1, ..., 1}
c
      do 67 i=1, n
      tempo1(i) = 1.d0
 67    continue
```

```
      call mtimesv(g, tempo1, rhs, maxnr1, maxnc1, m, n)
      endif
c
      write(6,*) 'user input, or randomly generated matrix G mxn = '
       do 63 i=1, m
       write(6,*) (g(i,j), j=1,n)
 63    continue
      write(6,*) '                              '
c
      write(6,*) 'user input: right-hand-side (rhs) vector mx1 = '
       do 92 i=1, m
       write(6,*) 'rhs(-) = ',rhs(i)
 92    continue
      write(6,*) '                              '
c
c
      call transpose(g, gt, maxnr1, maxnc1, m, n)
c
c-------------------------------------------------------
      if (m .ge. n) then
      call mtimesm(gt, g, am, maxnr1, maxnc1, maxnc2, n, m, n)  ! compute G' * G
      neq = n
c
      write(6,*) 'print [am] = G_transpose * G = '
      do 72 i=1,n
       write(6,*) (am(i,j),j=1,n)
 72   continue
      write(6,*) '                              '
c
      elseif (m .lt. n) then
      call mtimesm(g, gt, am, maxnr1, maxnc1, maxnc2, m, n, m)  ! compute G  * G'
      neq = m
c
      write(6,*) 'print [am] = G * G_transpose = '
      do 73 i=1,m
       write(6,*) (am(i,j),j=1,m)
 73   continue
      write(6,*) '                              '
c
      endif
c
c-------------------------------------------------------
c
      call generalized_inverse_cholesky(neq,am,idependrows,
     $ ndependrows,r, maxnr1, independrows, itempo1)
c
```

```
     write(6,*) 'special cholesky factor of Gt*G, or G*Gt  '
      do 22 i=1, r
       write(6,*) ( am(i,j), j=1,neq )
 22    continue
c
      write(6,*) '# independent rows = ',independrows
      write(6,*) '                            '

      if (independrows .ge. 1) then
      write(6,*) 'independent rows = ',(itempo1(i),i=1,independrows)
      endif
c
      call transpose(am, amt, maxnr1, maxnc1, r, neq)
      call mtimesm(am, amt, u, maxnr1, maxnc1, maxnc2, r, neq, r)  ! compute L' * L
      nactual = r
      call generalized_inverse_cholesky(nactual,u,idependrows,
     $ ndependrows,ractual, maxnr1, independrows, itempo1)

     write(6,*) 'regular cholesky factorization of M*Mt '
     write(6,*) 'M = factorized of Gt*G, or G*Gt with deleted rows'
      write(6,*) '# dependent rows = ndependrows = ',ndependrows
      write(6,*) '                            '

      if (iaxeqb .eq. 0) then     !  find generalized inverse explicitly

c
c......find the actual inverse of [u]  !!
c
      do 43 irow = 1, nactual
       do 42 i=1, nactual
       tempo1(i) = 0.d0
 42     continue
      tempo1(irow) = 1.d0
      call fbe_cholesky(nactual, u, tempo1, maxnr1)
c
       do 44 i=1, nactual
       am_inv(i,irow) = tempo1(i)
c       write(6,*) 'i=row#, irow=col#, am_inv(-,-) = ',i,irow,tempo1(i)
 44     continue
c
 43    continue
c
c......applying the French's generalized inverse formula
c
      if (m .ge. n) then
      call mtimesm(amt, am_inv, u, maxnr1, maxnc1, maxnc2, neq,
     $ nactual, nactual)                              ! compute L  * [am_inv]
```

```
      call mtimesm(u, am_inv, ut, maxnr1, maxnc1, maxnc2, neq,
     $ nactual, nactual)                     ! compute L  * [am_inv] * [am_inv]
      call mtimesm(ut, am, u, maxnr1, maxnc1, maxnc2, neq,
     $ nactual, neq)                         ! compute L  * [am_inv] * [am_inv] * L'
      call mtimesm(u, gt, ut, maxnr1, maxnc1, maxnc2, neq,
     $ neq, m )                              ! compute L  * [am_inv] * [am_inv] * L' * G'
       write(6,*) 'generalized inverse of [G] = '
       do 52 i=1, neq
c       write(6,*) (ut(i,j), j=1,m)
 52    continue
c
      elseif (m .lt. n) then
      call mtimesm(gt, amt, u, maxnr1, maxnc1, maxnc2, n, m,
     $ nactual)                              ! compute G' * L
      call mtimesm(u, am_inv, ut, maxnr1, maxnc1, maxnc2, n,
     $ nactual, nactual)                     ! compute G' * L  * [am_inv]
      call mtimesm(ut, am_inv, u, maxnr1, maxnc1, maxnc2, n,
     $ nactual, nactual)                     ! compute G' * L  * [am_inv] * [am_inv]
      call mtimesm(u, am, ut, maxnr1, maxnc1, maxnc2, n,
     $ nactual, neq)                         ! compute G' * L  * [am_inv] * [am_inv] *
L'
       write(6,*) 'generalized inverse of [G] = '
       do 54 i=1, n
        write(6,*) (ut(i,j), j=1,m)
 54    continue
      write(6,*) '                    '
      endif
c
c......solution of [G ] {x} = {rhs}
c           mxn nx1   mx1
c
c......thus,  {x} = [G+] * {rhs}
c         nx1   nxm   mx1
c
      call mtimesv(ut, rhs, tempo1, maxnr1, maxnc1, n, m)

      elseif (iaxeqb .eq. 1) then    !  AVOID computing generalized inverse explicitly

       if (m .ge. n) then
c......compute [G'] * {rhs}; with results stored in {tempo1}
      call mtimesv(gt, rhs, tempo1, maxnr1, maxnc1, n, m)
c     write(6,*) 'Gt * rhs = tempo1 = ',(tempo1(i),i=1,n)
c......compute [L'] * {tempo1}; with results stored in {tempo2}
      call mtimesv(am, tempo1, tempo2, maxnr1, maxnc1, nactual, n)
c     write(6,*) 'Gt * rhs = tempo2 = ',(tempo2(i),i=1,n)
c......now, doing forward & backward solutions, stored the results in {tempo2}
      call fbe_cholesky(nactual, u, tempo2, maxnr1)
```

```
c......now, doing forward & backward solutions AGAIN, stored the results in {tempo2}
      call fbe_cholesky(nactual, u, tempo2, maxnr1)
c......finally, compute [L] * {tempo2} = same as compute [G+] * {rhs}  !!
      call mtimesv(amt, tempo2, tempo1, maxnr1, maxnc1, n, nactual)

       elseif (m .lt. n) then
c......compute [L'] * {rhs}; with results stored in {tempo1}
      call mtimesv(am, rhs, tempo1, maxnr1, maxnc1, nactual, n)
c......now, doing forward & backward solutions, stored the results in {tempo1}
      call fbe_cholesky(nactual, u, tempo1, maxnr1)
c......now, doing forward & backward solutions AGAIN, stored the results in {tempo1}
      call fbe_cholesky(nactual, u, tempo1, maxnr1)
c......compute [L] * {tempo1}; with results stored in {tempo2}
      call mtimesv(amt, tempo1, tempo2, maxnr1, maxnc1, n, nactual)
c......finally, compute [G'] * {tempo2} = same as compute [G+] * {rhs}  !!
      call mtimesv(gt, tempo2, tempo1, maxnr1, maxnc1, n, m)
       endif
      endif
c
c......output unknown solution vector {x} with 3 numbers:
c......smallest (dabs{x}), biggest (dabs{x}), sum (dabs{x})
c
      abs_smallest = 10**8
      abs_biggest = 0.d0
      sum_abs = 0.d0

      write(6,*) 'solution vector {x} = pinv(G) * {rhs} is ...'
      do 102 i=1, n
       aa = dabs( tempo1(i) )
       bb = tempo1(i)
       write(6,*) 'i, x(i) = ',i, bb
       if (aa .LT. abs_smallest) abs_smallest = aa
       if (aa .GT. abs_biggest) abs_biggest = aa
       sum_abs = sum_abs + aa
 102   continue
      write(6,*) '                    '

      write(6,*) 'abs_smallest, abs_biggest, sum_abs = '
      write(6,*)  abs_smallest, abs_biggest, sum_abs
      write(6,*) '                    '
c
c......output absolute and relative error norm
c
      call error_norm(g,tempo1,rhs,maxnr1,maxnc1,m,n,abserr,relerr,
     $ tempo2)

      write(6,*) 'abserr, relerr = ',abserr, relerr
```

```
c
      call pierrotime(t2)

      time_a_to_z = t2 - t1

      write(6,*) 'time_a_to_z = ', time_a_to_z
     write(6,*) '                         '
c
c
     stop
     end
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
     subroutine generalized_inverse_cholesky(n,u,idependrows,
   $ ndependrows,r, maxnr1, independrows, independentrows)
     implicit real*8(a-h,o-z)
     integer r, ractual
     dimension u(maxnr1,*),idependrows(*),independentrows(*)
c=====================================================================
===============
c......Remarks:
c     (a)  Identifying which are dependent rows of a "floating" substructure
c     (b)  Factorizing (by cholesky) of floating substructure stiffness
c          Whenever a dependent row is encountered during cholesky process,
c          then we just:
c          [1] set all factorized values of the dependent row to be ZEROES
c          [2] ignore the dependent row(s) in all future factorized rows
c     (c)  [K "float"] = [K11]  [K12]
c                        [K21]  [K22]
c          where [K11] = full rank ( = non-singular )
c                      = {K "float"] with deleting the dependent rows/columns
c     (d)  The U_transpose * U of [K11] can be obtained by taking the results
c          of part (b) and deleting the dependent rows/columns
c......Author(s):  Prof. Duc T. Nguyen
c......Version:    02-11-2012
c......Stored at:  cd ~/cee/*odu*clas*/generalized_inverse_by_ldl.f
c=====================================================================
===============
c
c     write(6,*) 'check point #01'

     eps=0.0000000001
     ndependrows = 0
     independrows = 0
c
     r = 0

     do 11 ir = 1, n
```

```
        r = r + 1
c       write(6,*) ' ir, r = ', ir, r

        do 12 icol=ir, n
         sum = u(ir, icol)

          do 13 iprevrow=1, r-1
           sum = sum - u(iprevrow,ir) * u(iprevrow,icol)
 13        continue

c       write(6,*) 'check point #02'

        if (ir .eq. icol) then
c...... cholesky factorized diagonal terms of row # ir

c       write(6,*) 'check point #03'

          if (sum .gt. eps) then
           u(r,ir)=dsqrt(sum)
c           write(6,*) ' u(r,ir) = ',u(r,ir)
           independrows = independrows + 1
           independentrows(independrows) = ir
c       write(6,*) 'check point #04'
          else
           ndependrows = ndependrows + 1
c           idependrows(ndependrows) = ir
c           write(6,*) 'sum, u(r,ir) = diag term are ... ',sum,u(r,ir)
           r = r - 1
c       write(6,*) ' ir, r = ', ir, r
c       write(6,*) 'check point #05'
           go to 11
          endif
c
        else
c...... cholesky factorized off-diagonal terms of row # ir
c       write(6,*) 'check point #06'
          u(r,icol) = sum/u(r,ir)
        endif
c
 12     continue
c
 11     continue
c
c......all lower triangular of cholesky factorized [U] are set to zero !
c
```

```
      do 22 ir=1, r
      independr = independentrows(ir)
       do 23 icol=1, independr-1
 23     u(ir,icol)=0.d0
 22    continue
c
c      write(6,*) 'ndependrows = ', ndependrows
c      write(6,*) 'dependent rows = ',(idependrows(i), i=1,ndependrows)
c
      return
      end
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      subroutine transpose(a, at, maxnr1, maxnc1, nr1, nc1)
      implicit real*8(a-h,o-z)
      dimension a(maxnr1,*), at(maxnc1,*)
c
      do 1 i=1, nr1
       do 2 j=1, nc1
       at(j,i) = a(i,j)
 2     continue
 1    continue
      return
      end
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      subroutine mtimesm(a, b, c, maxnr1, maxnc1, maxnc2, nr1,nc1,nc2)
      implicit real*8(a-h,o-z)
      dimension a(maxnr1,*), b(maxnc1,*), c(maxnr1,*)
c
      do 1 j=1, nc2
       do 2 i=1, nr1
       c(i,j) = 0.d0
        do 3 k=1, nc1
        c(i,j) = c(i,j) + a(i,k) * b(k,j)
 3      continue
 2     continue
 1    continue
      return
      end
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      subroutine fbe_cholesky(n, u, rhs, maxnr1)
      implicit real*8(a-h, o-z)
c
      dimension u(maxnr1,*), rhs(*)
c
```

```
c
c......forward cholesky solution
c
      do 1 j = 1, n
      sum = rhs(j)
       do 2 i=1, j-1
       sum = sum - u(i,j) * rhs(i)
 2      continue
       rhs(j) = sum/u(j,j)
 1     continue
c
c......backward cholesky solution
c
      do 4 j=n, 1, -1
      sum = rhs(j)
       do 5 i=j+1, n
       sum = sum - u(j,i) * rhs(i)
 5      continue
       rhs(j) = sum/u(j,j)
 4     continue
c
      return
      end
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      subroutine mtimesv(g, tempo1, rhs, maxnr1, maxnc1, m, n)
      implicit real*8(a-h,o-z)
c
      dimension g(maxnr1,*), tempo1(*), rhs(*)
c
      do 1 i=1, m
      sum = 0.d0
       do 2 j=1, n
       sum = sum + g(i,j) * tempo1(j)
 2      continue
      rhs(i) = sum
 1     continue
c
      return
      end
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      subroutine error_norm(a,x,b,maxnr1,maxnc1,nr1,nc1,abserr,relerr,
     $ tempo1)
      implicit real*8(a-h,o-z)
      dimension a(maxnr1,*), x(*), b(*), tempo1(*)
c
```

```
      call mtimesv(a, x, tempo1, maxnr1, maxnc1, nr1, nc1)
c
       abserr = 0.d0
       relerr = 0.d0
c
      do 1 i=1, nr1
      x(i) = tempo1(i) - b(i)
      abserr = abserr + x(i)**2
      relerr = relerr + b(i)**2
 1     continue

       abserr = dabs(abserr)
       relerr = dabs(relerr)
       relerr = abserr/relerr
c
      return
      end
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
      subroutine pierrotime (time)
      real tar(2)
      real*8 time

c     ...............................................................
c      purpose :
c       This routine returns the user + system execution time
c       The argument tar returns user time in the first element and
c       system time in the second element.  The function value is the
c       sum of user and system time.  This value approximates the
c       program's elapsed time on a quiet system.
c
c       Uncomment for your corresponding platform
c
c       Note: On the SGI the resolution of etime is  1/HZ
c
c     Output
c       time: user+sytem executime time
c     ...............................................................

c      SUN -Solaris
      time=etime(tar)

c      HP - HPUX
c      time=etime_(tar)          !f90
c      time=etime_(tar)          !f77

c      COMPAQ - alpha
```

```
c     time=etime(tar)

c     CRAY
c     time=tsecnd()

c     IBM
c     time=0.01*mclock()

c     SGI origin
c     time=etime(tar)

      return
      end
c%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%
```

C.1: A Complete Input File of an Educational FORTRAN Source Code of "Generalized Inverse" Algorithms

For SLE

```
7   7   0   4   1

1.d0   2.d0   -3.d0   2.d0   -2.d0   -3.d0   -2.d0
2.d0   4.d0   -6.d0   4.d0   -4.d0   -6.d0   -4.d0
-3.d0  -6.d0   9.d0   -6.d0   6.d0    9.d0    6.d0
2.d0   4.d0   -6.d0   5.d0   -1.d0   -5.d0   -7.d0
-2.d0  -4.d0   6.d0   -1.d0   13.d0   9.d0   -5.d0
-3.d0  -6.d0   9.d0   -5.d0   9.d0    13.d0   9.d0
-2.d0  -4.d0   6.d0   -7.d0   -5.d0   9.d0    27.d0
 -2.d0  -4.d0   6.d0   -7.d0   -5.d0   9.d0    27.d0
```

Note: MATLAB solution = (0, 0, 0, 0, 0, 0, 1) = satisfy SLE

C.2: A Complete Output File of an Educational FORTRAN Source Code of "Generalized Inverse" Algorithms

For SLE

```
==========================================================

today date: 04-23-2012; Prof. Duc T. Nguyen
==========================================================

user input: m,n,iautodata,irankn,iaxeqb =
7 7 0 4 1
```

user input, or randomly generated matrix G mxn =
1.0 2.0 -3.0 2.0 -2.0 -3.0 -2.0
2.0 4.0 -6.0 4.0 -4.0 -6.0 -4.0
-3.0 -6.0 9.0 -6.0 6.0 9.0 6.0
2.0 4.0 -6.0 5.0 -1.0 -5.0 -7.0
-2.0 -4.0 6.0 -1.0 13.0 9.0 -5.0
-3.0 -6.0 9.0 -5.0 9.0 13.0 9.0
-2.0 -4.0 6.0 -7.0 -5.0 9.0 27.0

user input: right-hand-side (rhs) vector mx1 =
rhs(-) = -2.0
rhs(-) = -4.0
rhs(-) = 6.0
rhs(-) = -7.0
rhs(-) = -5.0
rhs(-) = 9.0
rhs(-) = 27.0

print [am] = G_transpose * G =
35.0 70.0 -105.0 69.0 -73.0 -127.0 -113.0
70.0 140.0 -210.0 138.0 -146.0 -254.0 -226.0
-105.0 -210.0 315.0 -207.0 219.0 381.0 339.0
69.0 138.0 -207.0 156.0 -84.0 -246.0 -320.0
-73.0 -146.0 219.0 -84.0 332.0 278.0 -56.0
-127.0 -254.0 381.0 -246.0 278.0 482.0 434.0
-113.0 -226.0 339.0 -320.0 -56.0 434.0 940.0

special cholesky factor of Gt*G, or G*Gt
5.916079783099616 11.832159566199232 -17.74823934929885 11.663128715253528
-12.339252119036342 -21.46691807010432 -19.100486156864473
0.0E+0 0.0E+0 0.0E+0 4.468940430507951 13.406821291523839
0.9781800942313469
-21.756515429211084
0.0E+0 0.0E+0 0.0E+0 0.0E+0 0.0E+0 4.496064087029694 10.065074253426936
0.0E+0 0.0E+0 0.0E+0 0.0E+0 0.0E+0 0.0E+0 0.7209335773362233
independent rows = 4

independent rows = 1 4 6 7
regular cholesky factorization of M*Mt
M = factorized of Gt*G, or G*Gt with deleted rows
# dependent rows = ndependrows = 0

solution vector {x} = pinv(G) * {rhs} is ...
i, x(i) = 1 1.6874571068360796E-13
i, x(i) = 2 3.3749142136721593E-13
i, x(i) = 3 -5.06237132050824E-13

i, x(i) = 4 -2.8787848147707946E-13
i, x(i) = 5 -2.2136011299001E-12
i, x(i) = 6 2.915937886998499E-12
i, x(i) = 7 0.9999999999987078

abs_smallest, abs_biggest, sum_abs =
1.6874571068360796E-13 0.9999999999987078 1.0000000000051376

abserr, relerr = 8.793202261721907E-25 9.354470491193518E-28
time_a_to_z = 9.539998136460781E-4

APPENDIX D

MatlabMPI SOURCE CODE FOR MATRIX-MATRIX MULTIPLICATION (MATRIX IN DENSE FORMAT)

```
%Initialize MPI
MPI_Init;

%Create communicator
comm = MPI_COMM_WORLD;

%Get Size and Rank
comm_size = MPI_Comm_size(comm); %numtasks
my_rank = MPI_Comm_rank(comm);   %taskid


nn = 1000; % Matrix size

%tStart = tic;

% Master Processor task
if (my_rank==0)

% a = 10*rand(nn);
% b = 10*rand(nn);

for i = 1: nn
   for j = 1:nn
      a(i,j) = (i-1)+(j-1);
   end
end
a;

for i = 1: nn
   for j = 1:nn
      b(i,j) = (i+1)*(j+1);
   end
end
b;

Z = zeros(nn);
tStart = tic;

domains = comm_size-1; % numworkers
```

```
%divide matrix "b" to parts (domains)
len = floor(length(b)/domains);


  for i = 1:domains-1

    MPI_Send(i,1,comm,a(:,:),b(:,((i-1)*len)+1:i*len)) %send parts of matrices to slaves
    sent_part = sprintf('%g', i)
    i;
    disp('*******');
  end

  MPI_Send(domains, 1,comm,a(:,:),b(:,((domains-1)*len)+1:length(b))); %last part to slave
  disp('last part sent');
  disp('*******');



  for i = 1:domains
    Z = MPI_Recv(i,100,comm);
    Z;
    size(Z);
    recv = sprintf('%g', i);
  end

end %end master


if my_rank > 0 %slave

  [matrix_a matrix_b] = MPI_Recv(0,1,comm);

  %Computation

  [ra ca] = size(matrix_a);
  [rb cb] = size(matrix_b);


  for k = 1: cb
    for i = 1:ra
      c(i,k) = 0;
      for j = 1:ca
        c(i,k) = c(i,k) + matrix_a(i,j) * matrix_b(j,k);
      end
    end
  end
```

```
    Z = c;

    MPI_Send(0,100,comm,Z);
    exit;

end %slave

MPI_Finalize;
tElapsed = toc(tStart)
disp('Success');
```

APPENDIX E

GRAPHICAL COMPARISONS (IN TERMS OF COMPUTATIONAL TIMES) OF ODU-GINVERSE WITH OTHER ALGORITHMS

In this appendix, we graphically compare the computational times of ODU-ginverse with other existing algorithms. The description of the test problems can be found in section 5.

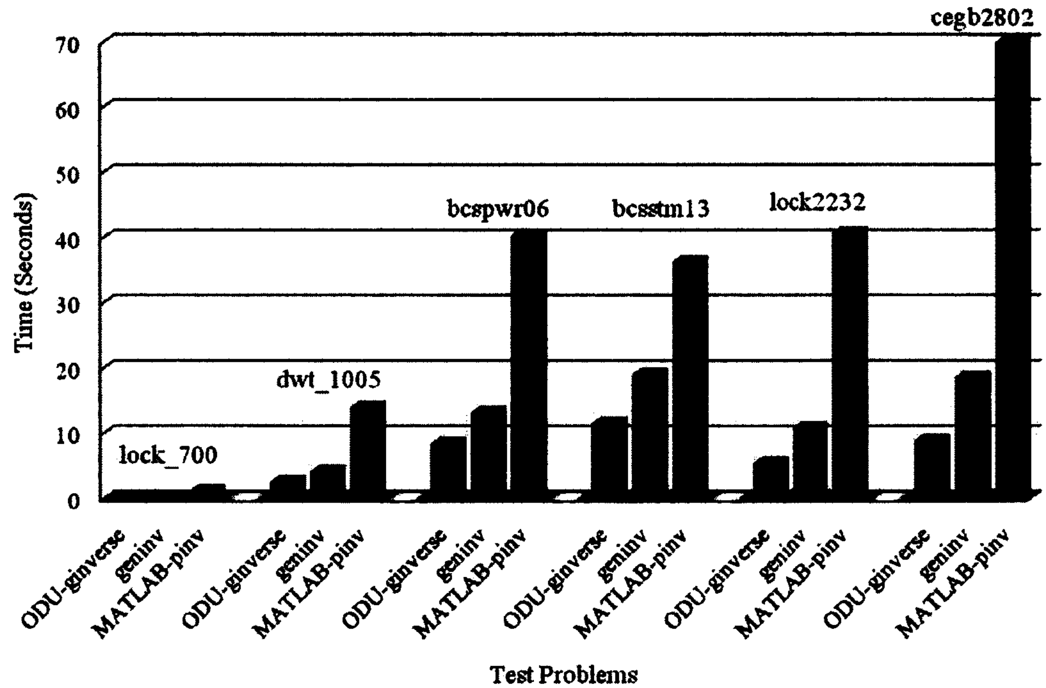

Figure E.1 Computational Times (in seconds) for Symmetric Rank-Deficient Test Matrices with RHS Vector as Linear Combination of Columns of Coefficient Matrix

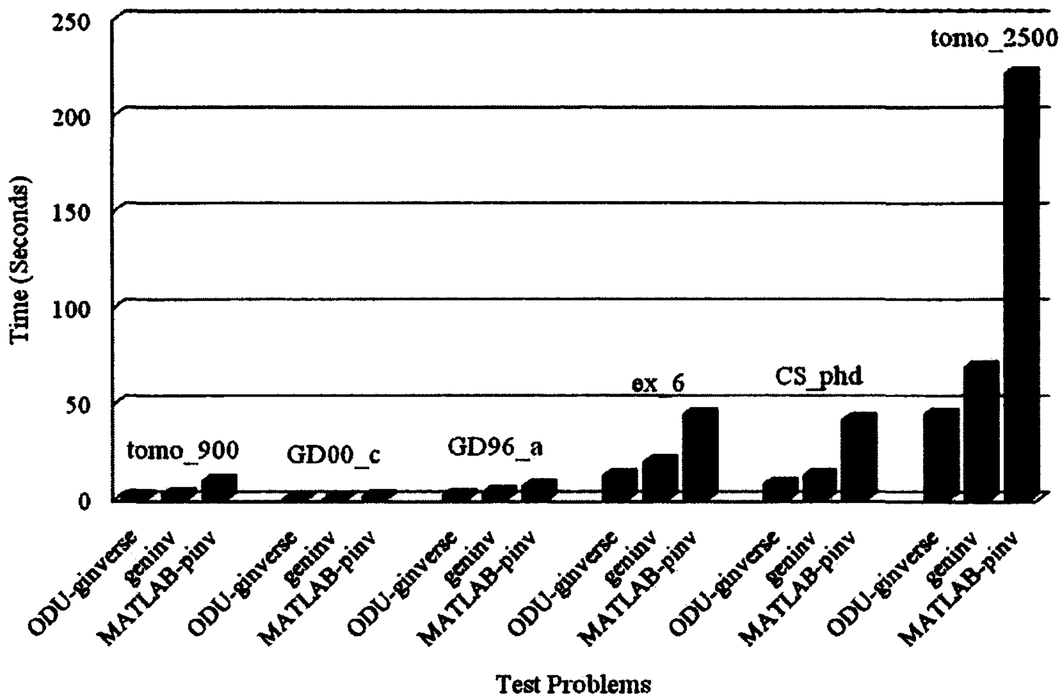

Figure E.2 Computational Times (in seconds) for Non-Symmetric Rank-Deficient Test Matrices with RHS Vector as Linear Combination of Columns of Coefficient Matrix

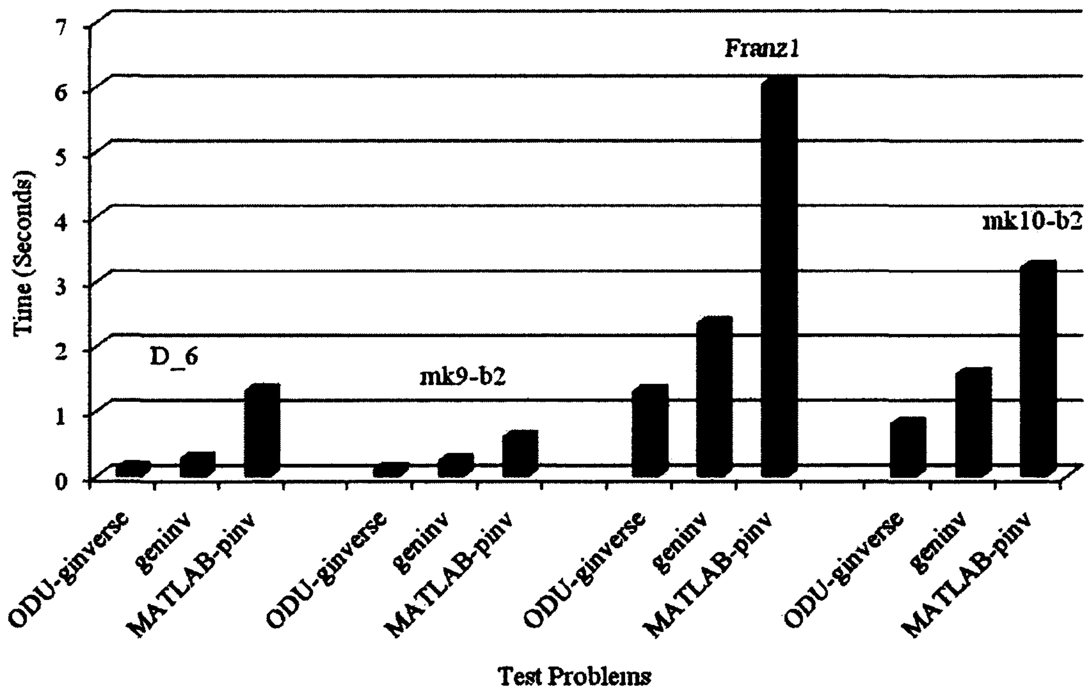

Figure E.3 Computational Times (in seconds) for Rectangular Rank-Deficient Test Matrices (tall type) with RHS Vector as Linear Combination of Columns of Coefficient Matrix

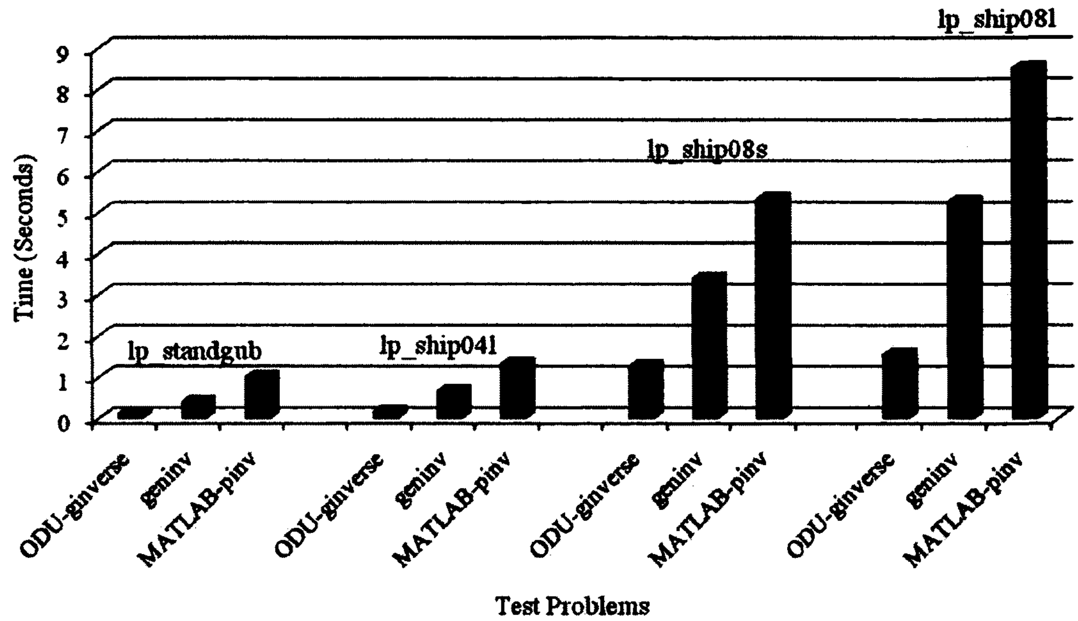

Figure E.4 Computational Times (in seconds) for Rectangular Rank-Deficient Test Matrices (fat type) with RHS Vector as Linear Combination of Columns of Coefficient Matrix

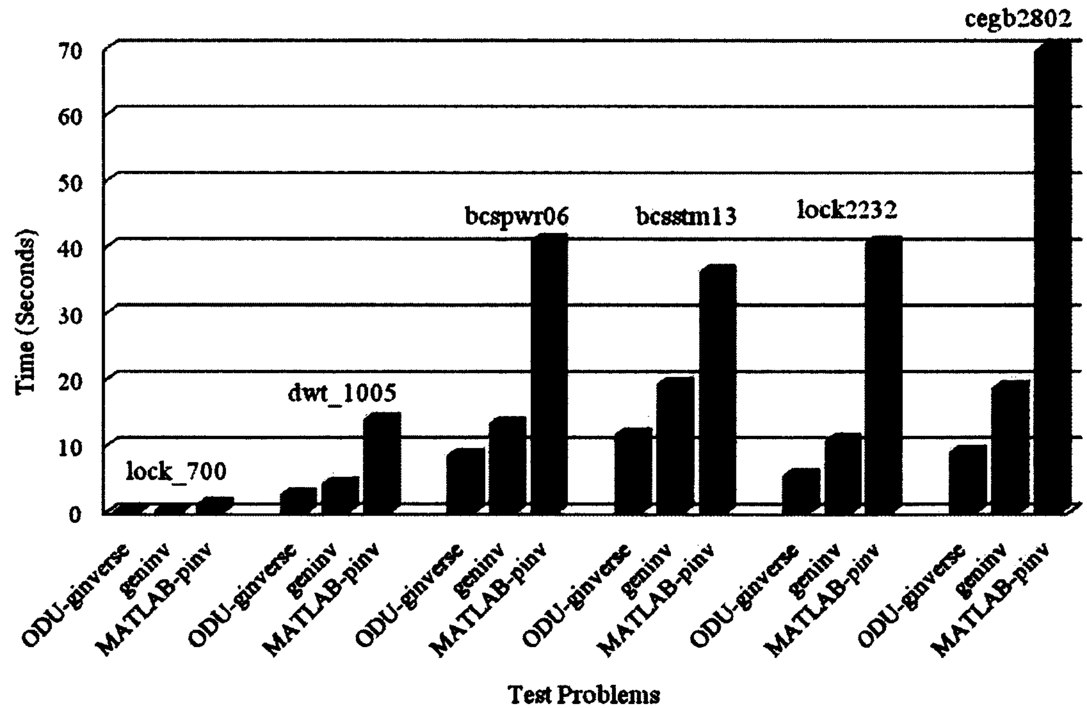

Figure E.5 Computational Times (in seconds) for Symmetric Rank-Deficient Test Matrices with Randomly generated RHS Vector

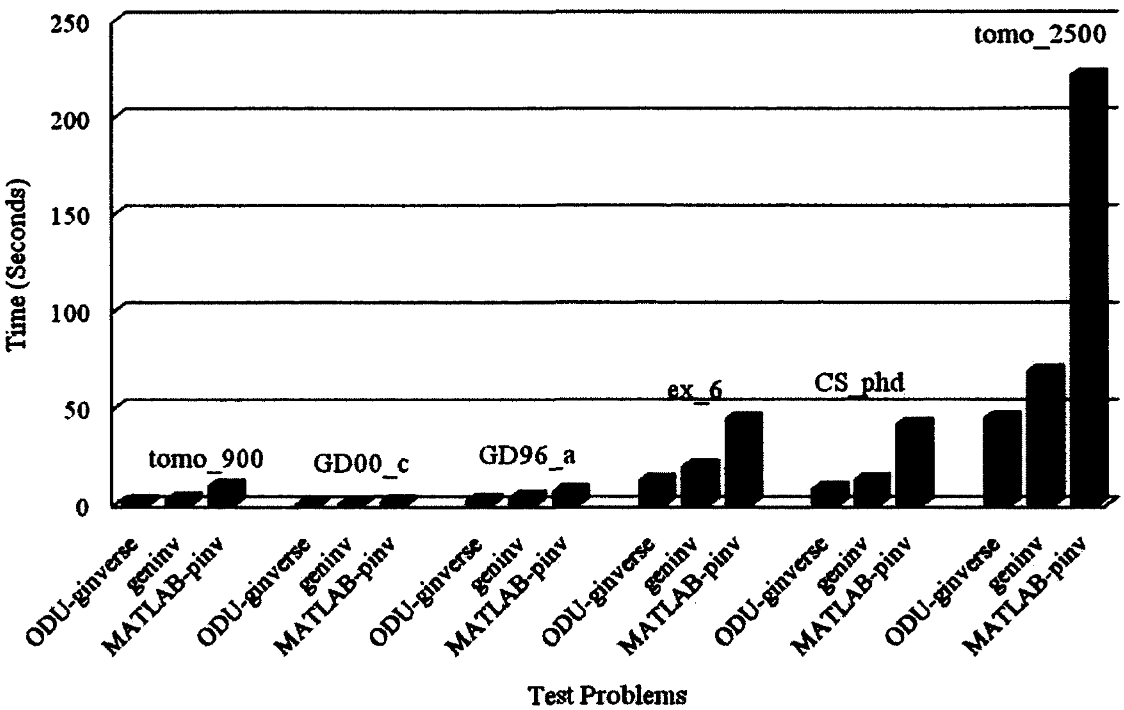

Figure E.6 Computational Times (in seconds) for Non-Symmetric Rank-Deficient Test Matrices with Randomly generated RHS Vector

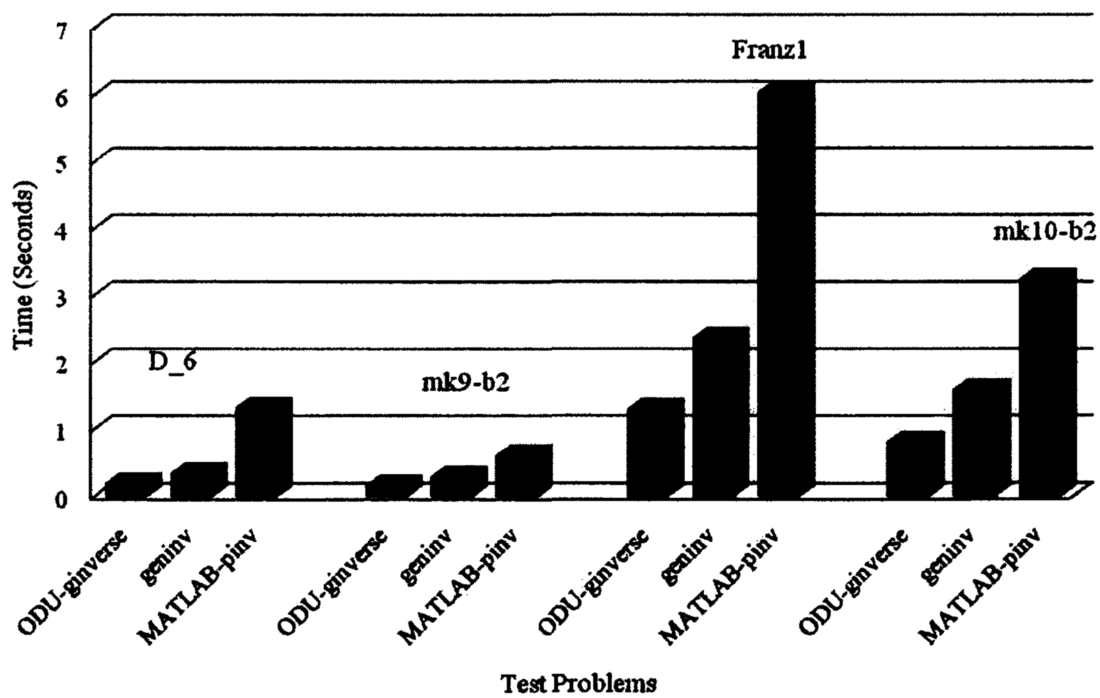

Figure E.7 Computational Times (in seconds) for Rectangular Rank-Deficient Test Matrices (tall type) with Randomly generated RHS Vector

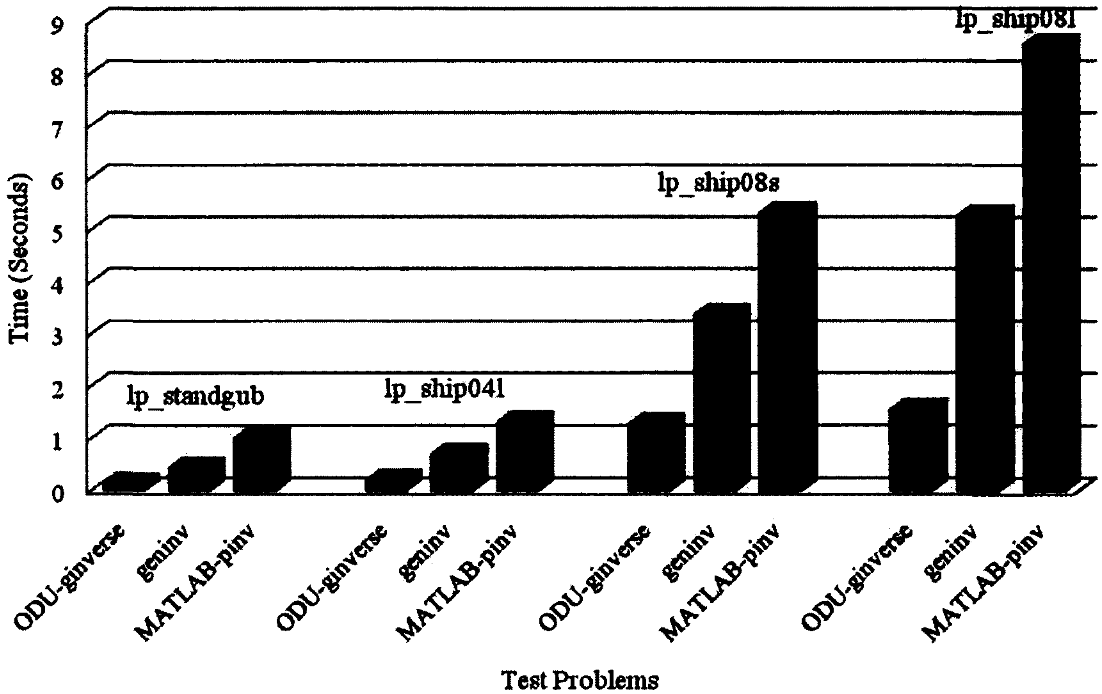

Figure E.8 Computational Times (in seconds) for Rectangular Rank-Deficient Test Matrices (fat type) with Randomly generated RHS Vector

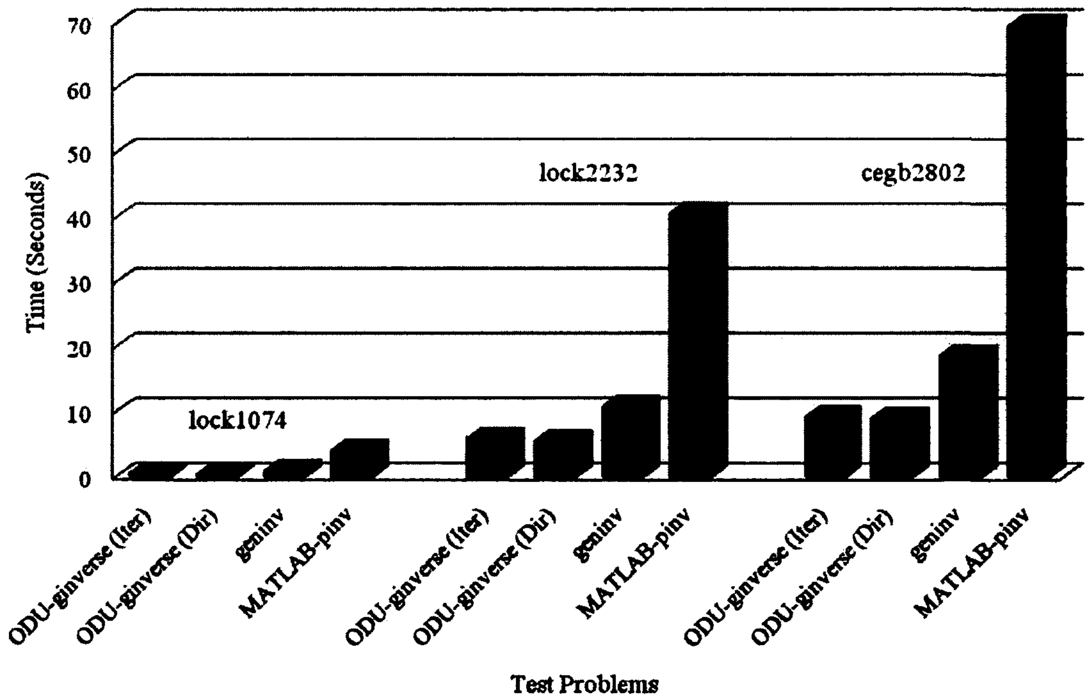

Figure E.9 Computational Times (in seconds) for Symmetric Rank-Deficient Test Matrices with Randomly generated RHS Vector (Iterative Solver inside Generalized Inverse)

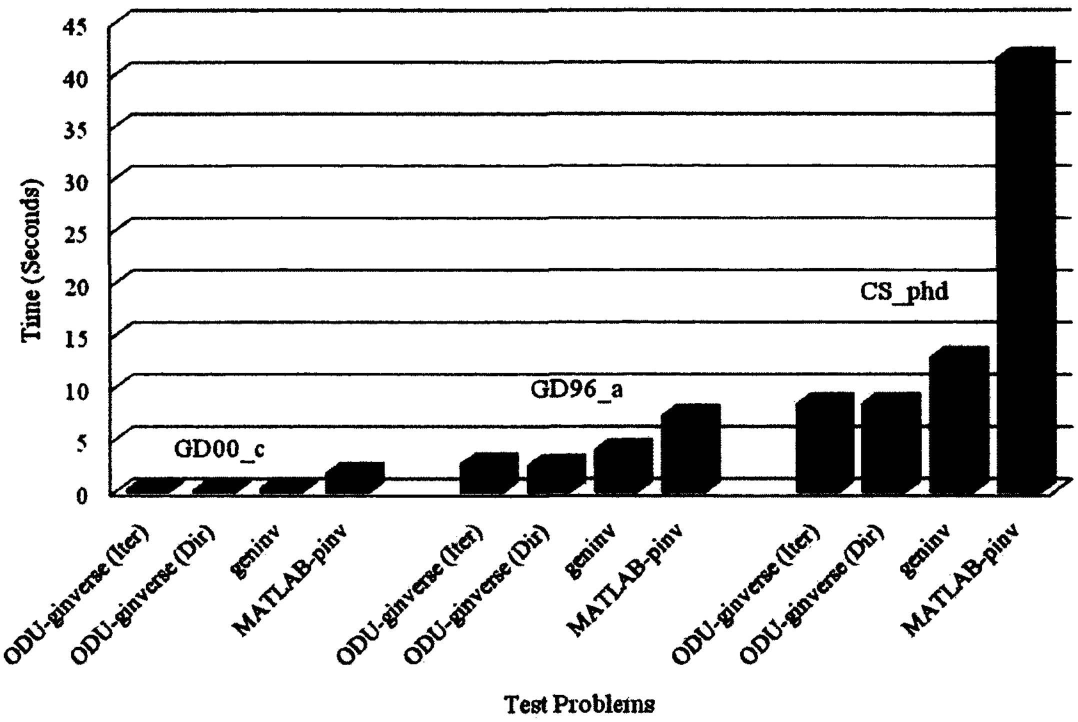

Figure E.10 Computational Times (in seconds) for Non-Symmetric Rank-Deficient Test Matrices with Randomly generated RHS Vector (Iterative Solver inside Generalized Inverse)

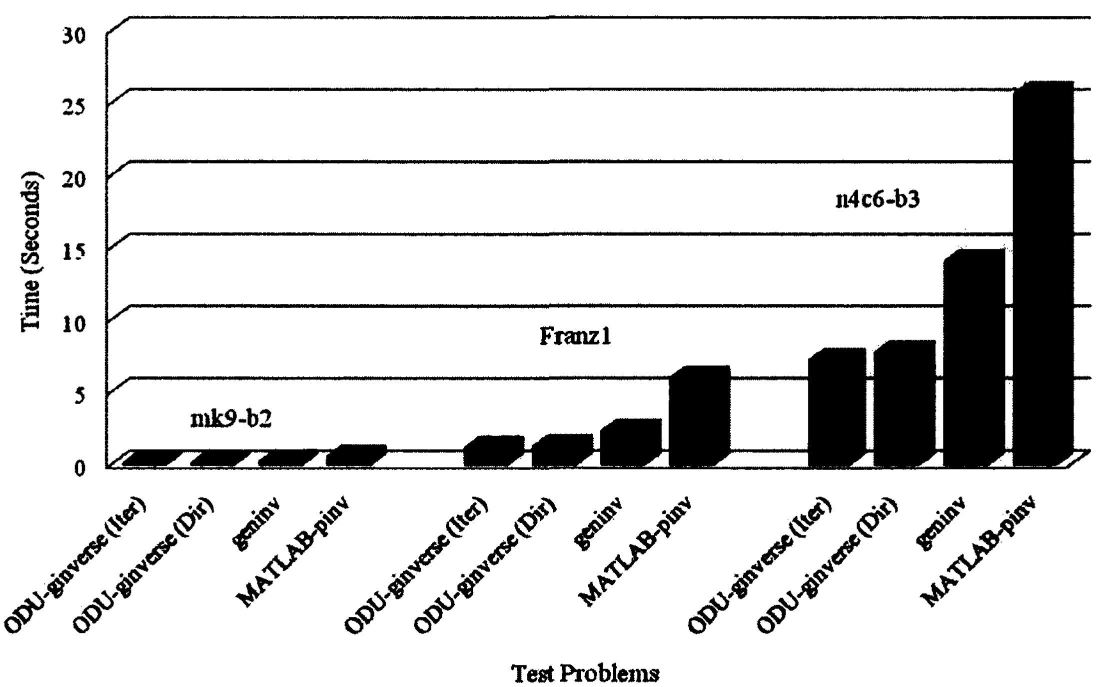

Figure E.11 Computational Times (in seconds) for Rectangular Rank-Deficient Test Matrices (tall) with Randomly generated RHS Vector (Iterative Solver inside Generalized Inverse)

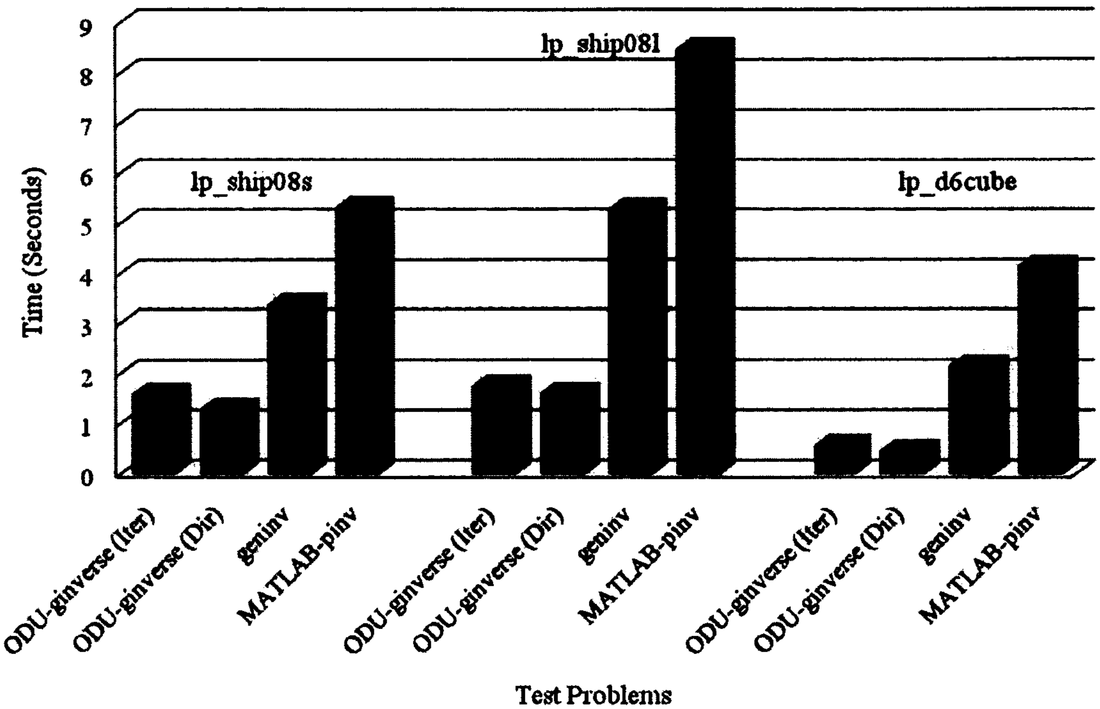

Figure E.12 Computational Times (in seconds) for Rectangular Rank-Deficient Test Matrices (fat) with Randomly generated RHS Vector (Iterative Solver inside Generalized Inverse)

VITAE

Education
Ph.D., Civil and Environmental Engineering, (Structural Engineering), Old Dominion University, Norfolk, Virginia 23529

Modeling and Simulation for Large Scale Computational Mechanics Certificate, Jan 2008 – Dec 2008, Civil and Environmental Engineering Department, Old Dominion University

Master of Technology, Structural Engineering, Acharya Nagarjuna University
Andhra Pradesh, India, May 2006

Bachelor of Engineering, Civil Engineering, Osmania University, Andhra Pradesh, India, June 2004

Research Interests
Numerical Methods, Parallel Programming, Algorithm Development, Finite Element Analysis, Advanced Structural Analysis, Engineering Optimization