

Old Dominion University

ODU Digital Commons

Civil & Environmental Engineering Theses & Dissertations

Civil & Environmental Engineering

Fall 2019

Parallel Jacobi Transformation Algorithm for Generalized Eigen-Solution With Improved Damage Detection of Truss/Bridge-Type Structures

Maryam Ehsaei

Old Dominion University, mehsa001@odu.edu

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



Part of the [Civil Engineering Commons](#)

Recommended Citation

Ehsaei, Maryam. "Parallel Jacobi Transformation Algorithm for Generalized Eigen-Solution With Improved Damage Detection of Truss/Bridge-Type Structures" (2019). Doctor of Philosophy (PhD), Dissertation, Civil & Environmental Engineering, Old Dominion University, DOI: 10.25777/7yj7-ww61
https://digitalcommons.odu.edu/cee_etds/102

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

**PARALLEL JACOBI TRANSFORMATION ALGORITHM FOR GENERALIZED EIGEN-SOLUTION WITH
IMPROVED DAMAGE DETECTION OF TRUSS/BRIDGE-TYPE STRUCTURES**

by

Maryam Ehsaei

B.S. September 2011, Fasa University, Iran

M.S. September 2013, Shiraz University, Iran

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

DOCTOR OF PHILOSOPHY

CIVIL AND ENVIRONMENTAL ENGINEERING

OLD DOMINION UNIVERSITY

October 2019

Approved by:

Duc T. Nguyen (Director)

Gene Hou (Member)

Yunbyeong Chae (Member)

Mojtaba Sirjani (Member)

ABSTRACT

PARALLEL JACOBI TRANSFORMATION ALGORITHM FOR GENERALIZED EIGEN-SOLUTION WITH IMPROVED DAMAGE DETECTION OF TRUSS/BRIDGE-TYPE STRUCTURES

Maryam Ehsaei

Old Dominion University, 2019

Director: Dr. Duc T. Nguyen

Serial Jacobi transformation algorithm for the solution of “standard eigen-problems” is re-visited to facilitate the explanation of the proposed parallel transformation algorithm, for which computational efficiency can be realized in this study through “pattern recognition” for the development and explanation of “explicit formulas” to avoid costly matrix time matrix operations. The proposed parallel Jacobi transformation for the solution of “generalized eigen-problems” has also been incorporated into the “improved damage detection” algorithm. Computational efficiency and robust behaviors for the entire proposed procedures (eigen-solution, damage detection and damage quantification) can be validated through several academic and real-life numerical examples. Numerical results obtained from this study have indicated that our proposed generalized Jacobi transformation is more robust/reliable as compared to MATLAB eigen-solver. Furthermore, our proposed simple rule of thumb for damage detection of aging bridge structures also give better results than existing algorithms.

Copyright, 2019, by Maryam Ehsaei, All Rights Reserved.

This thesis is dedicated to the proposition
that the harder you work, the luckier you get.

ACKNOWLEDGMENTS

There are a lot of people who contributed to this dissertation. First, I would like to thank my committee members, whose precious guidance and comments improved the quality of this manuscript. I would like to express my appreciation to my major advisor, Professor Nguyen, who devoted his precious time and knowledge guiding me and walking along with me through this journey; I highly appreciate his tireless efforts and devotion. Furthermore, I do thank my friends who were by my side during my program and this dissertation. Finally, I would like to thank my family, who has been my greatest support all my life. I devote this work to my father and mother, who sacrificed their lives to create mine in the best way possible, also to my sister, whose support and energy have been my greatest strengths.

NOMENCLATURE

K	Stiffness Matrix
M	Mass Matrix
λ	Eigen-Value Matrix
ϕ	Eigen-Vector Matrix
P_i	Rotation or Transformation Matrix
ω	Frequency Matrix
\tilde{F}_D	Flexibility Matrix
$E_i^{(e)}$	Strain Energy
K_R	Reduced Stiffness Matrix
M_R	Reduced Mass Matrix
$d_L^{(e)}$	Local element displacement
$d_G^{(e)}$	Global element displacement
$\bar{E}^{(e)}$	Normalized Cumulative Energy

TABLE OF CONTENTS

	Page
LIST OF TABLES.....	ix
LIST OF FIGURES.....	x
CHAPTER	
1. INTRODUCTION.....	1
1.1 LITERATURE SURVEYS.....	2
1.2 GOALS FOR THIS STUDY	3
1.3 ASSUMPTIONS FOR THIS STUDY	5
2. CLASSICAL JACOBI TRANSFORMATION AND THE GENERALIZED EIGEN PROBLEM.....	7
2.1 A REVIEW OF JACOBI TRANSFORMATION FOR THE SOLUTION OF THE “GENERALIZED EIGEN-PROBLEM”	8
2.2 Development OF “EXPLICIT FORMULAS” FOR TRIPLE MATRIX TIMES MATRIX OPERATIONS	12
2.3 PARALLEL COMPUTING STRATEGIES FOR JACOBI TRANSFORMATION ALGORITHM	19
2.4 SUBSPACE ITERATION	21
2.5 NUMERICAL EXAMPLES FOR SUBSPACE ITERATION WITH JACOBI ROTATION (PSI-JT) FOR EIGEN PROBLEM	23
3. EXISTING DAMAGE DETECTION AND NEW/PROPOSED ALGORITHMS.....	29
3.1 PHASE 1/2: DETECT/IDENTIFY THE DAMAGE MEMBERS.....	30
3.2 PHASE 2/2: DETERMINE THE LEVEL OF SEVERITY FOR THOSE FEW DAMAGED MEMBERS.....	36
3.3 NUMERICAL EXAMPLES FOR DAMAGE DETECTION AND DAMAGE QUANTIFICATION	39
4. CONCLUSIONS.....	47
5. REFERENCES.....	49
APPENDICES	52
A APPENDIX 1.....	52
A.1 SUBSPACE SOURCE CODE WITH JACOBI ROTATION COMBINATION	52
A.2 SUBSPACE SOURCE CODE WITH MATLAB “EIG” BUILT-IN FUNCTION	55
A.3 JACOBI ROTATION SOURCE CODE	58
B APPENDIX 2.....	67
B.1 SYMMETRIC STIFFNESS MATRIX, MODULE OF AN OFFSHORE PLATFORM	67
B.2 SYMMETRIC STIFFNESS MATRIX, FLUID FLOW GENERALIZED EIGENVALUES.....	69

	Page
B.3 SYMMETRIC STIFFNESS MATRIX, BUCKLING OF HOT WASHER	70
B.4 SYMMETRIC STIFFNESS MATRIX, MODULE OF AN OFFSHORE	71
B.5 SYMMETRIC STIFFNESS MATRIX, TRANSFORMATION TOWER, LUMPED MASSES	74
C APPENDIX 3	76
C.1 TRUSS CREATION SOURCE CODE	76
D APPENDIX 4.....	85
VITA	88

LIST OF TABLES

Table	Page
1. Motivations/Objectives for This Research Work	4
2. <u>2003 x 2003</u> Size Fluid Flow eig Solution Time and Solution Accuracy	24
3. <u>1086 x 1086</u> Size Buckling of Hot Washer eig Solution Time and Solution Accuracy	25
4. <u>420 x 420</u> Size Lumped Mass eig Solution Time and Solution Accuracy	25
5. <u>153 x 153</u> Size Transmission Tower eig Solution Time and Solution Accuracy	26
6. Fluid Flow Generalized Eigenvalues, Symmetric Stiffness Matrix, First 66 Eigen-Pairs	27
7. Fluid Flow Generalized Eigenvalues, Symmetric Stiffness Matrix, First 77 Eigen-Pairs	27
8. Fluid Flow Generalized Eigenvalues, Symmetric Stiffness Matrix, First 100 Eigen-Pairs	28
9. Fluid Flow Generalized Eigenvalues, Symmetric Stiffness Matrix, First 130 Eigen-Pairs	28
10. 11-bar Truss Examples with Different Damage Elements (Case 1, 2 and 3)	41
11. 48-bar Truss Example with 5 Damage Elements (Case 4)	43
12. Computation Time in Parallel Performance for Larger Scale Problem(Case 5)	44
13. Larger Scale Truss Example with 5 Damage Elements.....	45
14. Symmetric Stiffness Matrix, Module of an Offshore Platform, Properties	68
15. Symmetric Stiffness Matrix, Fluid Flow Generalized Eigenvalues, Properties	70
16. Symmetric Stiffness Matrix, Buckling of Hot Washer, Properties	72
17. Symmetric Stiffness Matrix, Medium Test Problem, Lumped Masses, Properties	73
18. Symmetric Stiffness Matrix, Transformation Tower, Lumped Masses, Properties....	75

LIST OF FIGURES

Figure	Page
1. A 6-Node, 11-Member Two-Dimensional Truss Structure	29
2. MATLAB Result for 11-bar Truss with 3 Damaged Members (1, 5, 10).....	41
3. MATLAB Result for 11-bar Truss with 4 Damaged Members (1, 5, 7, 10)	42
4. MATLAB Result for 11-bar Truss with 5 Damaged Members (1, 3, 6, 7, 9).....	42
5. MATLAB Result for 48-bar Truss Damaged Members (5, 13, 20, 35, 37).....	43
6. MATLAB Result for Larger Truss Damaged Members (10, 37, 55, 529, 705)	45
7. Symmetric Stiffness Matrix, Module of an Offshore Platform	68
8. Symmetric Stiffness Matrix, Fluid Flow Generalized Eigenvalues	69
9. Symmetric Stiffness Matrix, Buckling of Hot Washer	71
10. Symmetric Stiffness Matrix, Medium Test Problem, Lumped Masses	73
11. Symmetric Stiffness Matrix, Transformation Tower, Lumped Masses, properties....	74

CHAPTER 1

INTRODUCTION

During the past decades, substantial research efforts have been devoted to the development of damage identification techniques for civil engineering structures with both simulation and experimental studies. Based on the comprehensive literature reviews [1–3], vibration-based damage identification (VBDI) approaches have been widely developed and become an important research topic in the fields of civil, mechanical and aerospace engineering. Model-based techniques, a class of VBDI approaches, can be utilized effectively for both damage localization and quantification. In the techniques, an analytical or a numerical model (e.g. finite element methods) is generally required to give eigen-solutions of the monitored structure. As a result, performing eigen analysis with computational efficiency becomes one of the important factors affecting the effectiveness of this kind of model-based techniques.

For an undamped vibrating structure with N degrees-of-freedom, the “generalized eigen-problem” [4-8] can be described by the following equation:

$$K_{N \times N} \phi = \lambda M_{N \times N} \phi \quad (1)$$

For solving the above “generalized” eigen-problem, efficient solutions, such as Subspace Iteration [4, 7], Lanczos algorithms [4, 6-8] have been well documented in the literature. It should also be

mentioned that if the above $N \times N$ “Mass” matrix $[M]$ becomes an Identity matrix [4], then the above “generalized” eigen-problem will be simplified to the “standard” eigen-problem:

$$K_{N \times N} \phi = \lambda \phi \quad (2)$$

In Eqs. (1-2), K , λ and ϕ represent the system “stiffness,” “eigen-values” and “eigen-vectors” matrices, respectively. The Jacobi transformation/rotation family of algorithms [4-8] basically transforms the standard/generalized eigen-problem into diagonal matrix for easily computing all eigen-pairs.

1.1 Literature Surveys

Many researchers [4, 6-8] have considered the classical Jacobi rotation algorithm to transform the symmetrical, “standard eigen-problem” into diagonal matrix with all eigen-values appeared on its diagonal locations. Sameh and other researchers have extended the above classical (Jacobi rotation) procedure into “parallel Jacobi” algorithm [9] by simply demanding several (instead of only one) off-diagonal terms be driven to zero in each transformation. In Sameh’s prior work [9], however, all eigen-pairs of the “standard eigen-problem” need to be computed.

Bathe and other researchers have incorporated the classical Jacobi transformation into the subspace iteration algorithm [4] so that only the first few (or all) eigen-pairs can be found for the

“generalized eigen-problem.” Using the subspace iteration algorithm, the “sparse” matrix operations can be easily exploited [4, 7, 10]. However, in Bathe’s prior works [4], only one (not multiple) off-diagonal term at a time can be driven to zero.

1.2 Goals for This Study

The goals and objectives for this work are not only to extend the capability of the “stand-alone, generalized eigen-solver” [as shown in Table 1], but also to incorporate the parallel generalized eigen-solver into practical (real-life) engineering applications such as structural health monitoring. In this present work, first, the Jacobi transformation algorithm is embedded inside the subspace iteration algorithm to calculate the generalized eigen-problem of the monitored structure.

To provide the effective computational procedure, a parallel computing strategy based on the idea of making several off-diagonal terms to be simultaneously driven to zero is used for the Jacobi transformation algorithm, which is called parallel subspace iteration and Jacobi transformation (PSI-JT) algorithm. Then, the PSI-JT algorithm is incorporated into a two-phase damage identification method to improving the quality of damage assessment results in terms of the accurate solution and computational time.

Finally, 2-D and 3-D truss/bridge-type structures are presented to validate the superior performance of the proposed damage identification approach.

Table 1. Motivations/Objectives for This Research Work

	Standard Eigen- Problem	Generalized Eigen- Problem	Parallel Computation ^{^^}	All Eigen- Pairs	Few Lowest Eigen-Pairs	Sparse	Dense
Sameh's early works	Yes	No	Yes	Yes	No	No	Yes
K.J. Bathe's early works	Yes	Yes	No	Yes	Yes	Some	Yes
This dissertatio n/work	Yes	Yes	Yes	Yes	Yes	Yes	Yes

^{^^} Several (not just one) off-diagonal terms can be driven to zero in each transformation

The remaining sections of this dissertation will be organized as follows. After the introduction section, the classical Jacobi transformation for the solution of the “generalized eigen-problem” is briefly reviewed in Section 2.1. Next, in Section 2.2, explicit formulas (based on observed pattern recognitions) for the triple products (matrix times matrix) operations are developed and explained. Parallel computing strategies are presented in Section 2.3, for which Sameh's prior publications will be presented in a fashion such that the “explicit formulas” developed in Section

2.2 can be fully incorporated. Subspace iteration algorithm is summarized in Section 2.4, so that only “few lowest eigen-pairs” specified by the user can be computed for the “generalized eigen-problem.” Section 2.4 also shows that the stand-alone “Jacobi transformation” algorithm (presented in Sections 2.2, and 2.3) are embedded inside the subspace iteration algorithm. In Section 2.5, the superior performance (in terms of reduction in wall-clock time) of the parallel PSI-JT algorithm is investigated by comparing to the well-established MATLAB built-in eigen-solver such as the EIG function.

Existing damage detection and damage quantification are discussed in Section 3.1 and 3.2, for which a “simple rule of thumb” is proposed in section 3.1 to improve the quality of damage detection in bridge structures. Additional several numerical examples are presented in Section 3.3 to validate our claim for “improving the quality of damage detection” as compared to recently published algorithms. Finally, conclusion and future research works are highlighted in Section 4.

1.3 Assumptions for This Study

The following assumptions are made in this work:

Assumption 1: Damage can be imposed on the structure by specifying the level of damage (in percentage) occurred in certain members (not occurred in certain joints). For example, if member #5 of a 2-D truss structure is damaged by 30 % (or 0.30), then every term of the 4x4 element stiffness matrix of the damage member

#5 can be computed based on the original (undamage) member #5 element stiffness matrix, to be multiplied by the adjustment factor 0.70 (= $1.00 - 0.30$).

Assumption 2: For practical applications, the few sensor-locations should be placed at certain optimal locations (or at certain optimal degree-of-freedoms). Only the frequencies and mode-shapes (or eigen-vectors) at these sensor-locations are measured, while the information on system stiffness and mass matrices of the damage structure are unavailable. Thus, in this work we have assumed that the $L \times L$ eigen-vectors of the damage structure at the sensor-locations can be converted (or transformed) into the “full” $N \times L$ eigen-vectors (where $L \ll N$) through any existing model reduction methods (such as Guyan reduction method, Dynamic reduction method, etc.), which utilize the available information on the original (undamage) system stiffness and mass matrices.

CHAPTER 2

CLASSICAL JACOBI TRANSFORMATION AND THE GENERALIZED EIGEN PROBLEM

In the well-documented (classical) Jacobi transformation method, the original “stiffness” matrix $[K]$ and “mass” matrix $[M]$ in Eq. (1) can be repeatedly transformed into diagonal matrices, $[K^*]$ and $[M^*]$ respectively, through the Jacobi transformation as shown in Eqs. (3-4)

$$[K_{N \times N}] [\phi] = [\lambda] [M_{N \times N}] [\phi]; \text{ K and M are symmetrical.} \quad (\text{Eq. 1, repeated})$$

$$K^* = P_1^T K P_1 \quad (3)$$

$$M^* = P_1^T M P_1 \quad (4)$$

and the rotation (or transformation) matrix $[P_1]$ can be defined as:

$$P_1^T = \begin{bmatrix} 1 & \theta_1 & 0 & 0 \\ \theta_2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5)$$

In Eq. (5), we have assumed that the new off-diagonal terms for matrix K^* at location $(p, q) = (1, 2)$ to be driven to zero through the transformation shown in Eqs. (3-4). θ_1 and θ_2 are the 2 unknowns, which can be solved by applying the following equations:


$$K_{12}^* = 0, \text{ and } M_{12}^* = 0 \quad (6)$$

2.1. A Review of Jacobi Transformation for The Solution of the “Generalized Eigen-Problem”

The following derivation is valid, when k_{12} is intended to become zero. For the general case, two unknowns should be placed in k_{ij} and k_{ji} locations.

$$K^* = P_1^T K P_1 \quad (7)$$

$$= \begin{bmatrix} 1 & \theta_1 & 0 & 0 \\ \theta_2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} k_{11} & k_{12} & k_{13} & k_{14} \\ k_{21} & k_{22} & k_{23} & k_{24} \\ k_{31} & k_{32} & k_{33} & k_{34} \\ k_{41} & k_{42} & k_{43} & k_{44} \end{bmatrix} \begin{bmatrix} 1 & \theta_2 & 0 & 0 \\ \theta_1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (8)$$



$$\begin{bmatrix} k_{11} + k_{12}\theta_1 & k_{11}\theta_2 + k_{12} & k_{13} & k_{14} \\ k_{21} + k_{22}\theta_1 & k_{21}\theta_2 + k_{22} & k_{23} & k_{24} \\ k_{31} + k_{32}\theta_1 & k_{31}\theta_2 + k_{32} & k_{33} & k_{34} \\ k_{41} + k_{42}\theta_1 & k_{41}\theta_2 + k_{42} & k_{43} & k_{44} \end{bmatrix}$$

After performing the triple products shown in Eq. (8), K^* is obtained as it is represented in equation (9).

$$K^* = \quad (9)$$

$(k_{11} + k_{12}\theta_1)$ $+ \theta_1(k_{21} + k_{22}\theta_1)$	$(k_{11}\theta_2 + k_{12})$ $+ \theta_1(k_{21}\theta_2 + k_{22})$	$k_{13} + \theta_1 k_{23}$	$k_{14} + \theta_1 k_{24}$
<i>sym.</i>	$\theta_2(k_{11}\theta_2 + k_{12})$ $+ (k_{21}\theta_2 + k_{22})$	$\theta_2 k_{13} + k_{23}$	$\theta_2 k_{14} + k_{24}$
<i>sym.</i>	<i>sym.</i>	k_{33}	k_{34}
<i>sym.</i>	<i>sym.</i>	<i>sym.</i>	k_{44}

$$\text{Thus, } K^*_{1,2} = 0 = (k_{11}\theta_2 + k_{12}) + \theta_1(k_{21}\theta_2 + k_{22}) \quad (10)$$

$$M^*_{1,2} = 0 = (M_{11}\theta_2 + M_{12}) + \theta_1(M_{21}\theta_2 + M_{22}) \quad (11)$$

From Eqs. (10) & (11):

$$\theta_1 = \frac{-(k_{11}\theta_2 + k_{12})}{(k_{21}\theta_2 + k_{22})} = \frac{-(M_{11}\theta_2 + M_{12})}{(M_{21}\theta_2 + M_{22})} \quad (12)$$

Hence θ_2 can be computed from equation (12), as shown in the following paragraph.

From Eq. (12), one obtains:

$$(k_{11}\theta_2 + k_{12})(M_{21}\theta_2 + M_{22}) = (k_{21}\theta_2 + k_{22})(M_{11}\theta_2 + M_{12}) \quad (13)$$

$$k_{11}M_{21}\theta_2^2 + (k_{11}M_{22}+k_{12}M_{21}) \theta_2 + (k_{12}M_{22}) = k_{21}M_{11}\theta_2^2 + (k_{21}M_{12}+k_{22}M_{11}) \theta_2 + (k_{22}M_{12})$$

$$(k_{11}M_{21} - k_{21}M_{11}) \theta_2^2 + (k_{11}M_{22}+k_{12}M_{21} - k_{21}M_{12} - k_{22}M_{11}) \theta_2 + (k_{12}M_{22} - k_{22}M_{12}) = 0 \quad (14)$$

The above Eq. (14) can be expressed as:

$$(A_1)\theta_2^2 + (B_1)\theta_2 + (C_1) = 0 \quad (15)$$

Hence,

$$\theta_2 = \frac{-B_1 \pm \sqrt{B_1^2 - 4A_1C_1}}{2A_1} \quad (\text{assuming } A_1 \neq 0) \quad (16)$$

In Eq. (16), if the denominator $A_1 = 0$; then from (Eq. (15)), one obtains:

$$\theta_2 = -C_1 / B_1 \quad (17)$$

Finally, θ_1 can be found from Eq. (12)

The sign in front of the SQRT of Eq. (16) will be based on the sign of \bar{k} , defined as below.

$$\bar{k} = B_1 = (k_{11}M_{22} + k_{12}M_{21} - k_{21}M_{12} - k_{22}M_{11}) \quad (18)$$

After computing θ_2 [see Eq. (16), or Eq. (17)], and θ_1 [see Eq. (12)], matrix P_1^T can be generated as shown below:

$$P_1^T = \begin{bmatrix} 1 & \theta_1 & 0 & 0 \\ \theta_2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (19)$$

In the following steps, “explicit formulas” for the modified / transformed matrix K^* and M^* should be developed ($K^* = P_1^T K P_1$, and $M^* = P_1^T M P_1$). In the transformed matrix K^* and M^* , it is assumed the selected off-diagonal terms ($k_{12} = k_{21}$, and $m_{12} = m_{21}$) should be driven to zero.

The above procedure will be repeated until all the off-diagonal terms become zero. Equation 20 shows this procedure [4].

$$P_N^T \dots P_2^T P_1^T K P_1 P_2 \dots P_N = K^* \quad (20)$$

In Eq. (20), the matrix K^* eventually becomes a diagonal (eigen-value) matrix, where N is the size of $K_{N \times N}$. Furthermore, Eigen-Vectors matrix can also be identified from Eq. (20) [4, 7]:

$$P_1 P_2 \dots P_N = \phi \quad (21)$$

Based on Ref. [9], more than one off-diagonal terms can be driven to zero, which will also be adopted in this work.

The most time-consuming part of the Jacobi Rotation procedure is the computation, which involves with repeated matrix times matrix operations.

$$P_N^T \dots P_2^T P_1^T K P_1 P_2 \dots P_N$$

The diagram illustrates the scope of operations for the matrix expression $P_N^T \dots P_2^T P_1^T K P_1 P_2 \dots P_N$. Three horizontal double-headed arrows are shown below the expression, each labeled with a word: 'First', 'Second', and 'Last'. The 'First' arrow is the shortest, spanning from the first P_1^T to the first P_1 . The 'Second' arrow is longer, spanning from the first P_2^T to the second P_2 . The 'Last' arrow is the longest, spanning from the first P_N^T to the last P_N .

In this work, however, expensive matrix times matrix operations can be avoided by recognizing the pattern of “explicit formulas” for $P_{i+1}^T K P_{i+1}$, which will be explained in greater details in the next section.

2.2 Development of “Explicit Formulas” For Triple Matrix Times Matrix Operations

We have observed that there are specific patterns in the result of $K^* = P_1^T K P_1$ [see Eq. (9)], which will be repeated in every step of the procedure.

First of all, it is observed that the changes in matrix K^* (as compared to matrix K) only happens in the terms associated with the related rows and columns of matrix K (i^{th} row and j^{th} column for the selected K_{ij} , which will become zero, after the Jacobi transformation step $K^* = P_1^T K P_1$ is completed).

For better explanation, assuming that K_{12} [or K_{pq} , where $p=1$, and $q=2$] is selected to become zero after the Jacobi transformation. For the pairs (p, q) , it can be defined:

- The “companion” row for “row p ” is “row q ,” and the “companion” row for “row q ” is “row p .”
- The “companion” column for “column p ” is “column q ,” and the “companion” column for “column q ” is “column p .”

Recalled Eqs. (7-8), P_1 can be defined as:

$$P_1 = \begin{bmatrix} 1 & \theta_2 & 0 & 0 \\ \theta_1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (22)$$

*Matrix K^** can be computed as follows, based on Eqs. (3, 4, 9).

$(k_{11} + k_{12}\theta_1)$ $+ \theta_1(k_{21} + k_{22}\theta_1)$	$(k_{11}\theta_2 + k_{12})$ $+ \theta_1(k_{21}\theta_2 + k_{22})$	$k_{13} + \theta_1 k_{23}$	$k_{14} + \theta_1 k_{24}$
<i>sym.</i>	$\theta_2(k_{11}\theta_2 + k_{12})$ $+ (k_{21}\theta_2 + k_{22})$	$\theta_2 k_{13} + k_{23}$	$\theta_2 k_{14} + k_{24}$
<i>sym.</i>	<i>sym.</i>	k_{33}	k_{34}
<i>sym.</i>	<i>sym.</i>	<i>sym.</i>	k_{44}

In general, it has been observed that the transformation of all the components of K matrix, can be categorized in 3 different types. In other words, each of the components of matrix K will be transformed based on one of these three types.

These three types or categories are observed to be dependent on the location of the component in the transformed matrix K_{ij}^* as shown in Eq. (9). It is also observed that the developed formula is independent of the location of selected K_{ij} (selected component to become zero). The “explicit” formulas for each term K_{ij}^* can be developed based on the observed patterns, as described in the following paragraphs.

1. Type 1: All K_{ij} terms, which none of the indexes are either $p=1$ or $q=2$ (such as K_{33} , K_{34} and K_{44}). In other words, all terms K_{ij}^* for which $i \neq p, q$, and $j \neq q, p$

These (type 1) terms will not change after the triple product matrix multiplications ($K^* = P_1^T K P_1$) and their values remain the same.

2. Type 2: All K_{ij} terms, which only one of the indexes are either $p=1$ or $q=2$ (such as K_{13} , K_{14} , K_{23} , and K_{24}). In other words, all terms K_{ij}^* for which $i = \text{either } p, \text{ or } q$ and $j \neq p$ and $j \neq q$. These K_{ij}^* terms can be computed based on the following “explicit” formula:

$$K_{ij}^* = K_{ij} + \theta_m * K \text{ (“companion” row for “row } i, \text{” } j)} \quad (23)$$

The subscript m of θ can be found by looking at the “companion” row for “row i ” of the rotation matrix P_1 . Based on the “explicit” formula shown in Eq. (23), we can compute:

$$K_{13}^* = K_{13} + \theta_m * K \text{ (“companion” row for “row } 1, \text{” } 3)}$$

$$K_{13}^* = K_{13} + \theta_m * K_{23} \quad (24)$$

Where the subscript m of θ can be found by looking at the “companion” row for “row $i = 1$ ” of the rotation matrix P_1 . In this case, “companion” row for “row $i = 1$ ” is row 2 (by referring to $p=1$ and $q=2$). Thus, by looking at row 2 of matrix P_1 , it can be easily identified that $\theta_m = \theta_1$. Hence,

$$K_{13}^* = K_{13} + \theta_1^* K_{23} \quad (25)$$

Similarly, we can compute:

$$K_{24}^* = K_{24} + \theta_m^* K_{14} \text{ ("companion" row for "row 2," 4)}$$

$$K_{24}^* = K_{24} + \theta_m^* K_{14} \quad (26)$$

where the subscript m of θ can be found by looking at the "companion" row for "row $i = 2$ " of the rotation matrix P_1 . In this case, "companion" row for "row $i = 2$ " is row 1. Thus, by looking at row 1 of matrix P_1 , it can be easily identified that $\theta_m = \theta_2$. Hence,

$$K_{24}^* = K_{24} + \theta_2^* K_{14} \quad (27)$$

3. Type 3: All K_{ij} terms, for which both indices are either $p=1$ or $q=2$ (such as K_{11} , K_{12} and K_{22}).

In other words, all terms K_{ij}^* for which $i = \text{either } p, \text{ or } q$ and $j = \text{either } p, \text{ or } q$.

These K_{ij}^* terms can be computed based on the following 2 steps:

Step 3.1: In this step, the “exact, same” procedure as explained in Type 2 is followed. For example,

$$K_{12}^* = K_{12} + \theta_m * K \text{ (“companion” row for “row 1,”2)}$$

$$K_{12}^* = K_{12}^{^^} + \theta_m * K_{22}^{^^} \quad (28)$$

Then, referring to row 2 of matrix P_1 , the proper subscript m for theta is obtained, hence

$$K_{12}^* = K_{12}^{^^} + \theta_1 * K_{22}^{^^} \quad (29)$$

Step 3.2: In this step $K_{12}^{^^}$, shown in Eq. (29), is replaced by the following formulas:

$$K_{12}^{^^} = K_{12} + \theta_r * K \text{ (1, “companion” column for “column 2”)} \quad (30)$$

$$K_{12}^{^^} = K_{12} + \theta_r * K_{11}$$

where, the subscript “r” of θ can be obtained by referring to column 1 of matrix P_1^T . Thus,

$$K_{12}^{^^} = K_{12} + \theta_2 * K_{11} \quad (31)$$

Similarly, Replacing $K_{22}^{^^}$, shown in Eq. (29), by the following formulas:

$$K_{22}^{^^} = K_{22} + \theta_s * K_{21} \quad (32)$$

$$K_{22}^{^^} = K_{22} + \theta_s * K_{21} \quad (33)$$

where, the subscript “s” of θ can be obtained by referring to column 1 of matrix P_1^T .

Thus:

$$K_{22}^{^^} = K_{22} + \theta_2 * K_{21} \quad (34)$$

Finally, substituting Eqs. (31, 34) into Eq. (29), one obtains

$$K_{12}^* = \{ K_{12} + \theta_2 * K_{11} \} + \theta_1 * \{ K_{22} + \theta_2 * K_{21} \} \quad (35)$$

Similar procedures can be used to compute K_{11}^* , and K_{22}^* for these type 3 terms of matrix $[K^*]$.

2.3 Parallel Computing Strategies for Jacobi Transformation Algorithm

Sameh presented an algorithm [9] that can zero-out several off-diagonal terms (row “p,” column “q”) simultaneously, for the “Standard NxN Eigen-Problem.” This idea can also be applied for the “Generalized NxN Eigen-Problem,” where p and q are sequences defined by Sameh [9], in which p & q can be swapped, so that p is less than q. The complete algorithm (to systematically identify all the off-diagonal locations (p, q) of matrix [K]) driven by Sameh is presented in his early work in detail [9] and can be conveniently summarized here, as follows:

a) For $k = 1, 2, \dots, m-1$ [where $m = n / 2$; and $k = \text{step \#}$]

$$q = m - k + 1, m - k + 2, \dots, n - k,$$

$$p = (2m - 2k + 1) - q, \quad \text{if} \quad m - k + 1 \leq q \leq 2m - 2k$$

$$p = (4m - 2k) - q, \quad \text{if} \quad 2m - 2k < q \leq 2m - k - 1$$

$$p = n, \quad \text{if} \quad 2m - k - 1 < q$$

b) For $k = m, m+1, \dots, 2m-1$

$$q = 4m - n - k, 4m - n - k + 1, \dots, 3m - k - 1,$$

$$p = n, \quad \text{if} \quad q < 2m - k + 1$$

$$p = (4m - 2k) - q, \quad \text{if} \quad 2m - k + 1 \leq q \leq 4m - 2k - 1$$

$$p = (6m - 2k - 1) - q, \quad \text{if} \quad 4m - 2k - 1 < q$$

Example 1: For a 4x4 matrix $[K]$; $n = 4$; $k = \text{step \#} = 1, 2, \dots, n-1 = 3$;

For each step $m = n/2 = 2$ off-diagonal terms are simultaneously driven to zero. Applying the above algorithm, the following steps are produced:

- step #1: $(p, q) = (1,2) \ \& \ (3,4)$
- step #2: $(p, q) = (2,4) \ \& \ (1,3)$
- step #3: $(p, q) = (1,4) \ \& \ (2,3)$

Example 2: For a 6x6 matrix $[K]$; $n = 6$; $k = \text{step \#} = 1, 2, \dots, n-1 = 5$ F

or each step $m = n/2 = 3$ off-diagonal terms are simultaneously driven to zero. Applying the above algorithm the pairs are as below:

- step #1: $(p, q) = (2,3), (1,4) \ \& \ (5,6)$
- step #2: $(p, q) = (1,2), (3,5) \ \& \ (4,6)$
- step #3: $(p, q) = (3,6), (2,4) \ \& \ (1,5)$
- step #4: $(p, q) = (2,6), (1,3) \ \& \ (4,5)$
- step #5: $(p, q) = (1,6), (2,5) \ \& \ (3,4)$

Extension of Ref. [9] for Parallel Jacobi Transformation of “Generalized Eigen-Problems” is described in the following part of this section.

The generalized eigen-equations, see Eq. (1), can be re-stated as

$$[K_{N \times N}] [\phi] = [\lambda] [M_{N \times N}] [\phi] \quad (36)$$

In eq (36), K is a Symmetrical Positive Definite (SPD) “stiffness” matrix.

$$K^* = P_1^T K P_1 \quad (3, \text{repeated})$$

$$M^* = P_1^T M P_1 \quad (4, \text{repeated})$$

Assuming the off-diagonal terms of matrix K^* and M^* , at locations $(p, q) = (1, 2)$ & $(p, q) = (3, 4)$, are intended to be driven to zero. Thus,

$$P_1^T = \begin{bmatrix} 1 & \theta_1 & 0 & 0 \\ \theta_2 & 1 & 0 & 0 \\ 0 & 0 & 1 & \theta_3 \\ 0 & 0 & \theta_4 & 1 \end{bmatrix} \quad (37)$$

The 4 unknowns θ_1 , θ_2 , θ_3 and θ_4 can be solved by employing 4 associated equations $K_{12}^* = 0 = K_{34}^* = M_{12}^* = M_{34}^*$, and using similar “explicit formulas” developed in Section 2.1 of this dissertation.

2.4 Subspace Iteration

Subspace Iteration and Lanczos Algorithms [4-8] have been used extensively in the engineering communities for solving the generalized eigen-problem

$$K_{N \times N} \phi = \lambda M_{N \times N} \phi \quad (36, \text{repeated})$$

The details of “Subspace Iteration” algorithm is presented in the following step-by-step procedure:

Step 1: Guess $[X_k]_{N \times L}$, where $L \ll N$ and $L \approx (2 \text{ to } 4) \times (\# \text{ lowest Eigen Pairs desired})$

Step 2: The following equation is developed

$$[K] \bar{X}_{k+1} = [M] X_k \quad (38)$$

The unknown, $[\bar{X}_{k+1}]$, can be solved by sparse equation solver [6-8, 10], where K and M are sparse (SPD = Symmetric Positive Definite) matrices.

Step 3: Reduced problem is created in this step by applying the result from previous step to original stiffness and mass matrices. The following “reduced” stiffness and “reduced” mass matrices are obtained:

$$[K_{Reduce}]_{L \times L} = [\bar{X}_{k+1}^T]_{L \times N} [K]_{N \times N} [\bar{X}_{k+1}]_{N \times L} \quad (39)$$

$$[M_{Reduce}]_{L \times L} = [\bar{X}_{k+1}^T]_{L \times N} [M]_{N \times N} [\bar{X}_{k+1}]_{N \times L}$$

Step 4: Solve for all eigen-pairs of the Generalized (Dense) Reduced Eigen-Problem [see Jacobi transformation with explicit formulas in Section 2.2]:

$$[K_R]_{L \times L} [E_Vectors]_{L \times L} = [E_Values]_{L \times L} [M_R]_{L \times L} [E_Vectors]_{L \times L} \quad (40)$$

Step 5: In this step the guessed (eigen-vector) matrix $[X]$ is being update using equation (41).

$$[X_{k+1}]_{N \times L} = [\bar{X}_{k+1}]_{N \times L} \times [E_Vectors]_{L \times L} \quad (41)$$

If the algorithm converges, then the subspace iteration process stops, if the algorithm is not converged, then, returns to Step 2, and replaces X_k by X_{k+1} . This procedure will continue until the convergence achieved [4].

2.5 Numerical Examples for Subspace Iteration with Jacobi Rotation (PSI-JT) for Eigen-Problems

In this section, several illustrative test examples are used to evaluate the performance of the proposed PSI-JT algorithm, in both MATLAB sequential and parallel computing environments. The results for eigen-solutions, and wall-clock time are reported in Tables 2-5.

All the examples are real world eigen value problems, which shows the PSI-JT algorithm super performance in comparison with MATLAB built-in function.

Table 2. 2003 x 2003 Size Fluid Flow eig Solution Time and Solution Accuracy

Requested Eigenvalue	PSI-JT algorithm	MATLAB “eig”
2	2.449043 (9 subspace iteration)	2.458226 (9 subspace iteration)
4	2.742210 (8 subspace iteration)	2.454659 (8 subspace iteration)
10	6.689142 (8 subspace iteration)	Not converged
27	102.968629 (8 subspace iteration)	Not converged
63	2027.442472 (7 subspace iteration)	Not converged

Table 3. 1086 x 1086 Size Buckling of Hot Washer eig Solution Time and Solution Accuracy

Requested Eigenvalue	PSI-JT algorithm	MATLAB “eig”
2	1.916372 (9 subspace iteration)	1.650959 (27 subspace iteration)
4	2.348291 (11 subspace iteration)	1.449550 (20 subspace iteration)
10	5.795341 (8 subspace iteration)	Not converged
27	98.628914 (8 subspace iteration)	Not converged
63	2016.937281 (7subspaceiteration)	Not converged

Table 4. 420 x 420 Size Lumped Mass eig Solution Time and Solution Accuracy

Requested Eigenvalue	PSI-JT algorithm	MATLAB “eig”
2	0.199795 (9 subspace iteration)	0.092030 (11 subspace iteration)
4	0.376640 (7 subspace iteration)	0.091801 (7 subspace iteration)
10	4.459468 (7 subspace iteration)	Not converged
20	1405.341492 (7 subspace iteration)	Not converged

Table 5. 153 x 153 Size Transmission Tower eig Solution Time and Solution Accuracy

Requested Eigenvalue	PSI-JT algorithm	MATLAB “eig”
2	0.078787 (4 subspace iteration)	0.010875 (4 subspace iteration)
4	0.316909 (7 subspace iteration)	0.020544 (9 subspace iteration)
10	3.417312 (8 subspace iteration)	Not converged
20	30.654872 (7 subspace iteration)	Not converged
27	1602.563036 (8 subspace iteration)	Not converged
28	2690.135469 (8 subspace iteration)	Not converged

To follow, a different number of eigen-pairs for a specific problem is requested. The parallel performance and time comparison for this example using different number of processors are represented in Tables 6-9. This example is a real-world fluid flow eigen-value problem, in which the stiffness matrix is a module of an offshore platform [Refs. 23]. The stiffness matrix has exactly 3948 rows and 3948 columns. It is a sparse, symmetric positive definite matrix that is a structural full rank matrix. A high number of components makes it time consuming for non-parallel algorithms to solve and order the eigen-pairs of such matrix. However, by using the proposed algorithm a few numbers of eigen-pairs can be found in a reasonable amount of time.

Table 6. Fluid Flow Generalized Eigenvalues, Symmetric Stiffness Matrix, First 66 Eigen-Pairs

Requested Eigenvalue	Entire subspace iteration 1 processor (sec)	1 processor time (sec)	Entire subspace iteration 2 processors (sec)	2 processor time (sec)	Entire subspace iteration 3 processors (sec)	3 processors time (sec)	Entire subspace iteration 4 processors (sec)	4 processors time (sec)
66	896.455950	895.378442	603.305860	602.345941	599.162143	598.197305	542.307915	541.302544
	1102.822512	1101.811450	730.748058	729.745724	725.859754	724.858474	667.134515	666.160063
	1261.487762	1260.467557	856.724612	855.684254	815.686974	814.699774	763.602743	762.586148
	895.840537	894.852254	608.52093	607.505687	591.797457	590.830891	541.680371	540.677359
	590.352192	589.355626	389.60349	388.605737	386.427996	385.435389	350.564300	349.563550
	422.939290	421.939288	280.34919	279.365551	281.492801	280.537490	254.537911	253.554994
Average	861.6497	860.6341	578.2087	577.2088	566.7379	565.7599	519.9713	518.9741
Speed Ratio		1		1.491027		1.5212		1.658337
Efficiency		100%		74.5%		50.7%		41.4%

Table 7. Fluid Flow Generalized Eigenvalues, Symmetric Stiffness Matrix, First 77 Eigen-Pairs

Requested Eigenvalue	Entire subspace iteration 1 processor (sec)	1 processor time (sec)	Entire subspace iteration 2 processors (sec)	2 processor time (sec)	Entire subspace iteration 3 processors (sec)	3 processors time (sec)	Entire subspace iteration 4 processors (sec)	4 processors time (sec)
77	831.959080	830.658220	713.884270	712.550378	681.975138	680.613784	606.455958	605.202129
	1051.925227	1050.620045	880.492157	879.018793	817.823681	816.488608	752.659846	751.401258
	1048.577007	1047.274977	865.129932	863.853816	863.766488	862.427766	770.183558	768.925017
	743.936809	742.619260	607.147950	605.873926	617.980968	616.640425	550.739695	549.483642
	523.077149	521.780760	426.449668	425.161826	415.184607	413.828006	390.863231	389.598018
	436.999096	435.712405	348.092683	346.781326	343.967879	342.706388	326.716534	325.461272
	393.059580	391.798155	334.649933	333.348337	309.103373	307.825279	293.608608	292.369932
Average	718.5048	717.2091	596.5495	595.2269	578.5432	577.2186	527.3182	526.063
Speed Ratio		1		1.204934		1.242526		1.363352
Efficiency		100%		60%		41.4%		34%

Table 8. Fluid Flow Generalized Eigenvalues, Symmetric Stiffness Matrix, First 100 Eigen-Pairs

Requested Eigenvalue	Entire subspace iteration 1 processor (sec)	1 processor time (sec)	Entire subspace iteration 2 processors (sec)	2 processor time (sec)	Entire subspace iteration 3 processors (sec)	3 processors time (sec)	Entire subspace iteration 4 processors (sec)	4 processors time (sec)
100	1789.793154	1788.015892	1183.647068	1182.113589	1089.22413	1087.59278	1057.477909	1055.889081
	2187.909222	2186.397163	1482.387773	1480.840859	1304.322926	1302.711819	1346.470352	1344.894885
	2011.821475	2010.273289	1363.752152	1362.213095	1320.355531	1318.70554	1222.838462	1221.248836
	1751.386294	1749.809894	1144.917301	1143.397802	1056.207567	1054.589811	1066.126696	1064.553007
	1235.837517	1234.150799	801.335118	799.791688	840.833167	839.214997	758.246017	756.671522
	982.869197	981.286765	631.880441	630.311019	679.717967	678.119807	586.747670	585.171944
	805.975200	804.441556	517.550627	515.988933	571.278204	569.617665	485.437057	483.870980
Average	1537.942	1536.339	1017.924	1016.38	980.2771	978.6503	931.9063	930.3286
Speed Ratio		1		1.51158		1.569855		1.651394
Efficiency		100%		75%		52%		41%

Table 9. Fluid Flow Generalized Eigenvalues, Symmetric Stiffness Matrix, First 130 Eigen-Pairs

Requested Eigenvalue	Entire subspace iteration 1 processor (sec)	1 processor time (sec)	Entire subspace iteration 2 processors (sec)	2 processor time (sec)	Entire subspace iteration 3 processors (sec)	3 processors time (sec)	Entire subspace iteration 4 processors (sec)	4 processors time (sec)
130	4674.504531	4672.470960	2914.733795	2912.711002	2798.485735	2796.439071	2694.347694	2692.321249
	5317.978785	5315.944539	3324.582042	3322.546426	3211.935270	3209.894113	3022.486226	3020.435974
	5758.926719	5756.906390	3607.435549	3605.413389	3525.006478	3522.958084	3302.339200	3300.250400
	4693.046095	4691.039332	2946.655990	2944.637561	2912.528066	2910.519497	2711.994936	2709.959192
	3862.506603	3860.490993	2458.863334	2456.850108	2474.853233	2472.807510	2289.757198	2287.742729
	3195.744506	3193.679225	2019.590105	2017.597980	2006.354640	2004.329649	1869.879441	1867.833520
	2561.290700	2559.251701	1633.518384	1631.485206	1656.405730	1654.390045	1523.149886	1521.121354
	2349.782457	2347.762727	1482.917552	1480.925914	1519.861581	1517.850097	1373.881566	1371.864847
	2138.589593	2136.548267	1360.015090	1358.008694	1359.840351	1357.800250	1276.555119	1274.509227
Average	3839.152	3837.122	2416.479	2414.464	2385.03	2382.999	2229.377	2227.338
Speed Ratio		1		1.589223		1.610207		1.722739
Efficiency		100%		79.5%		53.6%		43%

CHAPTER 3

EXISTING DAMAGE DETECTION AND NEW/PROPOSED ALGORITHMS

Damage detection in structures, specifically bridge type structures, is an important subject. Due to its important application in real world problems, this topic attracts a lot of old and new scholars to research on this topic. A lot of researchers have investigated damage detection or health monitoring problems and presented methods [11-18].

In this chapter, a two-phase method is presented for damage detection using a “simple rule of thumb” for structural damage detection and quantification. The merit of the present two-phase method over other exiting two-phase methods [13,14] is that a simple but efficient “rule of thumb” is proposed for the improvement in damage detection, together with the parallel PSI-JT algorithm that is incorporated to effectively compute for the generalized eigen-problem.

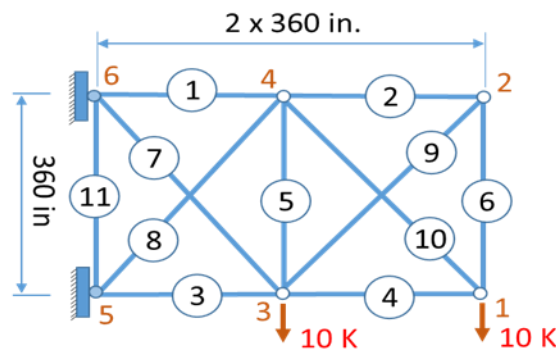


Figure 1. A 6-Node, 11-Member Two-Dimensional Truss Structure

To illustrate damage localization and quantification process of the two-phase method, an example of a 2-D Truss/Bridge Structure, shown in Figure 1, is used herein. In Figure 1, the lengths of each member, the cross-sectional area of each member, the material density and Young's modulus are user's input parameters. In general, the Finite Element Method (FEM) will be based on the type of structure we wish to analyze. This illustrative example is based on a 2-D Truss/Bridge type structure. Using FEM, a structure can be analyzed under (a) undamaged (original) condition, and (b) damaged condition.

Once the frequencies (related to eigen-values) & mode-shapes (eigen-vectors) of the damage structure is measured (via optimal locations of sensors), the proposed method can robustly detect the "location (Phase 1/2) and the severity (Phase 2/2)" of damage members. The step-by-step numerical procedures of this two-phase method can be summarized in the following sections.

3.1. Phase 1/2: Detect/Identify Damage Members

Step 1.0 Finite Element Analysis of "Original" (Undamage) Structure

In this step, first the element stiffness $[k_L^{(e)}]$ matrices, and the element diagonal/lumped mass $[m_L^{(e)}]$ matrices are computed.

Then, the global stiffness $[K] = \sum [k_G^{(e)}]$ matrix, and the global $[M] = \sum [m_G^{(e)}]$ diagonal/ lumped mass matrix is assembled. Using MATLAB command equation (42) is driven.

$$[\phi, \lambda] = EIG(K_{bc}, M_{bc}) \quad (42)$$

Then, the Eigen Values ($[\lambda]$ and frequencies, ω_i) can be obtained, and the corresponding Eigen Vectors (mode-shapes ϕ_i) can be identified through the matrix $[\phi]$. MATLAB “EIG” command will solve the “generalized” eigen-equation:

$$[K_{bc}]\phi_i^* = \omega_i^2 [M_{bc}] \phi_i^* \quad (43)$$

Next, the mass-orthonormalized scalar of each eigen vector is computed.

$$\{\phi_i^*\}^T [M_{bc}] \{\phi_i^*\} = \text{scalar} = c_i \quad (44)$$

$$\{\phi_i\} = \frac{\{\phi_i^*\}}{\sqrt{c_i}} \quad (45)$$

Thus,

$$\widetilde{F_{UD}} = \widetilde{F_{UnDamaged}} = \sum_{i=1}^{NLM} \frac{1}{\omega_i} \phi_i \phi_i^T; \text{ where NLM = Number of Lowest Modes} \quad (46)$$

Step 2.0 (very similar to Step 1.0)

Using FEM, the associated damaged structure is also analyzed. In real life structure, the measurements of frequencies & mode shapes would come from sensors installed on the structure at key locations. For our example, “artificial damage” is applied to elements #1, #5 and #10 of the mentioned example [see Figure 1], with stiffness reduction of 80%, 70% and 90% for those 3 elements, respectively.

In this step, it would be desirable to compute the element stiffness matrices $[k_L^{(e)}]$ with damage members. However, the element mass $[m_L^{(e)}]$ diagonal matrices with no damage applied is required to be used.

Next, the global damaged stiffness $[K] = \sum [k_G^{(e)}]$, and the global $[M] = \sum [m_G^{(e)}]$ diagonal lumped mass matrices are assembled respectively. Then, boundary conditions are applied on the system's stiffness and mass matrices $[K_{bc}]$ and $[M_{bc}]$, respectively. Using the MATLAB command `eig` represents in equation (47) the eigen pairs are obtained.

$$[\phi, \lambda] = \text{EIG}(K_{bc}, M_{bc}) \quad (47)$$

Then, the Eigen Values ($[\lambda]$ and frequencies ω_i) is obtained. The corresponding Eigen Vectors (mode- shapes ϕ_i) can be identified by the matrix $[\phi]$. MATLAB command `EIG` will solve the “generalized” eigen-equation represented in eq (48).

$$[K_{bc}]_D \phi_i^* = \omega_i^2 [M_{bc}] \phi_i^* \quad (48)$$

Remarks: After obtaining the eigen-solution for damage structure, it is pretended that the damage members and their severities are unknown.

Then, the Mass-Orthonormalized scalar of each eigen vector is computed.

$$\{\phi_i^*\}^T [M_{bc}] \{\phi_i^*\} = \text{scalar} = c_i \quad (49)$$

$$\{\phi_i\} = \frac{\{\phi_i^*\}}{\sqrt{c_i}} \text{ Thus,} \quad (50)$$

$$\tilde{F}_D = \tilde{F}_{Damaged} = \sum_{i=1}^{NLM} \frac{1}{\omega_i^2} \phi_i \phi_i^T \quad (51)$$

$$\tilde{F}_\Delta = \tilde{F}_{UD} - \tilde{F}_D \quad (52)$$

$$[U, S, V] = SVD(\tilde{F}_\Delta) \quad (53)$$

Then by using MATLAB “SVD” command, which is represented in eq (53). the given matrix [see Eq. (52)] into its triple products is decomposed, where the second (or middle) matrix is a diagonal matrix, and the first & third matrices are ORTHOGONAL matrices:

$$\tilde{F}_\Delta = [U][\varepsilon][V]^T \quad (54)$$

$$= [[U_1] \quad [U_0]] \begin{bmatrix} [\varepsilon_1] & [0] \\ [0] & [0] \end{bmatrix} [[V_1] \quad [V_0]]^T \quad (55)$$

“If the column vectors in the matrix $[V_0]$ are treated like different loading conditions/vectors [19, 20], then the stresses of damage elements will be equal to zero.” In practical application, we should use “Strain Energy” $E_i^{(e)}$, instead of stress associated with each e^{th} element, and check for low strain elements [13-14].

$$E_i^{(e)} = \frac{1}{2} \{d_L^{(e)}\}^T [k_L^{(e)}] d_L^{(e)} = \text{scalar}; \quad (56)$$

where $i = 1, 2, 3, \dots$ ndlv = number of damaged location vectors = # of columns of the sub-matrix $[V_0]$.

Notes: the above elements’ strain energy is associated with the “original (undamage)” structure, since the goal of Phase 1 is to find and identify “which members are damaged.”

$$\{d_L^{(e)}\} = [R^{(e)}] d_G^{(e)} \quad (57)$$

where:

$$[R^{(e)}] = \begin{bmatrix} C_x & S_x & 0 & 0 \\ -S_x & C_x & 0 & 0 \\ 0 & 0 & C_x & S_x \\ 0 & 0 & -S_x & C_x \end{bmatrix}; \text{ and } C_x = \frac{x_j - x_i}{L^{(e)}}; \quad C_y = \frac{y_j - y_i}{L^{(e)}} \quad (58)$$

$$L^{(e)} = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2} \quad (59)$$

Instead of using Stress, or Strain Energy for each element, we use the “Normalized Cumulative Energy,” or “NCE” for each element [21,22], which is defined as

$$\bar{E}^{(e)} = \frac{\psi^{(e)}}{\max_k \{\psi^k\}} \quad (60)$$

$$\text{where, } \psi^{(e)} = \sum_{i=1}^{ndlv} \frac{E_i^{(e)}}{\max_k \{E_i^k\}} \quad (61)$$

For each e^{th} element (corresponding to the i^{th} loading case), $E_i^{(e)}$ is computed as shown above for the undamage case. Within each i^{th} loading case, the max value among all elements “e” is found and the $\max_k \{E_i^k\}$ is obtained. Then, Eq. (61) is applied to compute $\psi^{(e)}$.

Among all $\psi^{(e)}$ values computed, the max value = $\max_k \{\psi^k\}$ is found and Eq. (60) is applied to compute “NCE” $\equiv \bar{E}^{(e)}$. Based on the computed “NCE” = $\bar{E}^{(e)}$, eq (62) is obtained.

$$\bar{E}^{(e)} = \left\{ \begin{array}{c} \bar{E}^{(1)} \\ \bar{E}^{(2)} \\ \bar{E}^{(3)} \\ \bar{E}^{(4)} \\ \bar{E}^{(5)} \\ \bar{E}^{(6)} \\ \bar{E}^{(7)} \\ \bar{E}^{(8)} \\ \bar{E}^{(9)} \\ \bar{E}^{(10)} \\ \bar{E}^{(11)} \end{array} \right\} = \left\{ \begin{array}{c} \text{very close to zero (even when 3 or 4 lowest modes used)} \\ \bar{E}^{(2)} \\ \bar{E}^{(3)} \\ \bar{E}^{(4)} \\ \text{very close to zero (even when 3 or 4 lowest modes used)} \\ \bar{E}^{(6)} \\ \bar{E}^{(7)} \\ \bar{E}^{(8)} \\ \bar{E}^{(9)} \\ \text{very close to zero (when 6 lowest modes used)} \\ \text{exactly zero (Element \#11 is connected by 2 pinned nodes)} \end{array} \right\}$$

above formula.....(62)

Notice that $\bar{E}^{(11)}$ is exactly zero. However, element #11 should NOT be considered as a damage element, because this element has 2 end nodes which are fully constrained by 2 pinned (Dirichlet boundary condition) supports. This element has its nodal displacements equal to zero, thus it has no stress and has zero “normalized cumulative energy.”

3.2. Phase 2/2: Determine the Level of Severity for Those Few Damage Members

Using optimization techniques, such as Genetic Algorithm (GA), or Differential Evolution (DE), etc., one can find the level (or amount) of damage occurred in elements # (1), # (5) and # (10) that have already been found/identified in Phase 1/2.

Let \vec{x} = the unknown amount of damage in the truss elements # (1), # (5) and # (10).

$$\vec{x} = \begin{cases} x^{(1)} = [0.00 \rightarrow 1.00] \\ x^{(2)} = [0.00 \rightarrow 1.00] \\ x^{(3)} = [0.00 \rightarrow 1.00] \end{cases} \quad (63)$$

Thus, the optimization problem can be stated. The unknown vector \vec{x} is found, such that the OBJECTIVE function $\Gamma(\vec{x})$, defined in eq. (64), is minimized [22].

$$\text{Min. } \Gamma(\vec{x}) = 1 - MDLAC(\vec{x}) + \sum_{i=1}^{NLM} \frac{\|\phi_{DM,i} - \phi_{DA,i}(\vec{x})\|}{\|\phi_{DM,i}\|} \quad (64)$$

In eq. (64), $\phi_{DM,i}$ = the i^{th} damaged mode shape, which can be obtained by measurements (using sensors at strategic/optimal locations), in real-life/practical applications.

$\phi_{DA,i}(\vec{x})$ = the i^{th} analytical (damage) mode-shape, associated with the current amount of damage vector \vec{x} , found by the optimization (GA, or DE, etc...) process. In this dissertation example, the actual measurements have not been taken. Instead, artificially created damage conditions to VALIDATE the numerical procedures.

$$MDLAC(\vec{x}) = \frac{|\Delta f^T \delta f(\vec{x})|^2}{(\Delta f^T \Delta f)(\delta f^T(\vec{x}) \delta f(\vec{x}))} \leq 1 \quad (65)$$

The right-hand side of the above inequality can be easily verified by Cauchy's inequality, and

$$\Delta f = \frac{\|\vec{f}_{ud} - \vec{f}_{DM}\|}{\|\vec{f}_{ud}\|} \quad (66)$$

$$\delta f(\vec{x}) = \frac{\|\vec{f}_{ud} - \vec{f}_{DA}(\vec{x})\|}{\|\vec{f}_{ud}\|} \quad (67)$$

“If” $\vec{f}_{DM} = \vec{f}_{DA}(\vec{x})$, as the measured frequency vector of the damage structure is equal to the analytical (damage) frequency vector, “Then,” the Eqs. (66-67) will lead to $\Delta f = \delta f(\vec{x})$, and Eq. (65) will become $MDLAC(\vec{x}) = 1$.

Hence the Minimum value for the objective function will become [see Eq. (64)]:

$$\text{Min. } \Gamma(\vec{x}) = 1 - [\text{MDLAC}(\vec{x}) = 1] + \{\text{summation term} = 0\} = 0$$

In this work, a “simple rule of thumb” has been added for improving damage detection phase. This rule of thumb basically states that “if the Normalized Cumulative Energy of an element is less than or equal to a specific factor, say 10 (based on our numerical experience) times $\min(\bar{E}^{(e)})$, then that member should also be considered as a damage element.” However, this “rule of thumb” should obviously NOT be applied for finding the minimum energy for any member with fully constraints at its end nodes, such as member 11 of Figure 1).

$$E^{(e)} \leq 10 \times \min(\bar{E}^{(e)}) \quad (68)$$

3.3. Numerical Examples for Damage Detection and Damage Quantification

In this Section, several numerical examples are used to evaluate the performance of the proposed “simple rule of thumb,” which basically modify the existing algorithms for Damage Detection and Damage Quantification of Bridge-type Structures.

Comparisons between existing algorithms [13, 14], and the proposed “simple/inexpensive rule of thumb” are reported in Tables 10, 11 and 13, and in Figures 2-5. All the figures are the last iteration results, which the meaning of each diagram is explained in follow.

In all figures, the upper diagram, X-axes show the “number of variations,” which represents the number of damage elements (for instance, the number of bars shows the number of damage elements), and Y-axes named as “current best individual” show the severity of damage elements for each of the damage members.

In the lower diagram, the X-axis shows “score” that indicates the fitness (objective) function value, and this Y-axis also shows number of populations, which falls within the score ranges.

It is worth mentioning that these figures have been created by MATLAB software automatically and represent the convergence of the problem to the results, which are shown in these figures. In other words, upper figure shows the number of damage members and their damage severities, and the lower figure shows the number of individuals and their respective fitness value range

(for example, in figure 2, almost 30 individuals in the population has the fitness value in range of $0.2-0.5 \times 10^{-3}$). Summation of all bars' heights in the lower diagram gives the population size generated by MATLAB code.

In this work, different sizes for 2-D and 3-D truss/bridge-type problems have been investigated, using the proposed algorithm. In each example, some elements are considered to be damaged with different levels of severity. It is shown in the following problems that the improved algorithm, can easily recognize the damage elements and their severities (either low or high), regardless of the input amount of severities on damage elements. It is worth mentioning that existing algorithms [13, 14] are unable to detect all of the damage members, especially those with low severity, in some cases, as it is fully described in the related papers [13, 14].

Table 10. 11-bar Truss Examples with Different Damage Elements (Case 1, 2 and 3)

3 damage elements			4 damage elements			5 damage elements		
Damage Element	Damage Severity	Detected elements by existed alg.	Damage Element	Damage Severity	Detected elements by existed alg.	Damage Element	Damage Severity	Detected elements by existed alg.
1	80%	detected	1	20%	Not detected	1	70%	detected
5	70%	detected	7	10%	Not detected	3	50%	Not detected
10	90%	detected	5	30%	Not detected	6	70%	detected
			10	50%	detected	7	20%	Not detected
						9	40%	Not detected

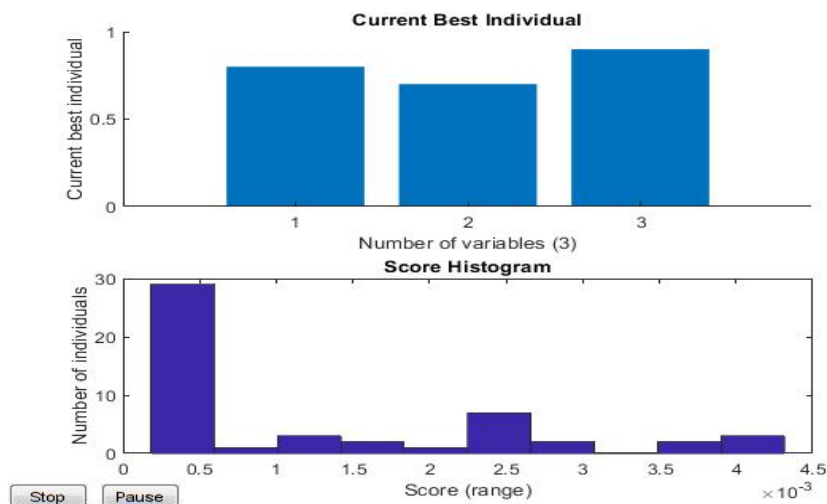


Figure 2. MATLAB Result for 11-bar Truss with 3 Damage Members (1, 5, 10)

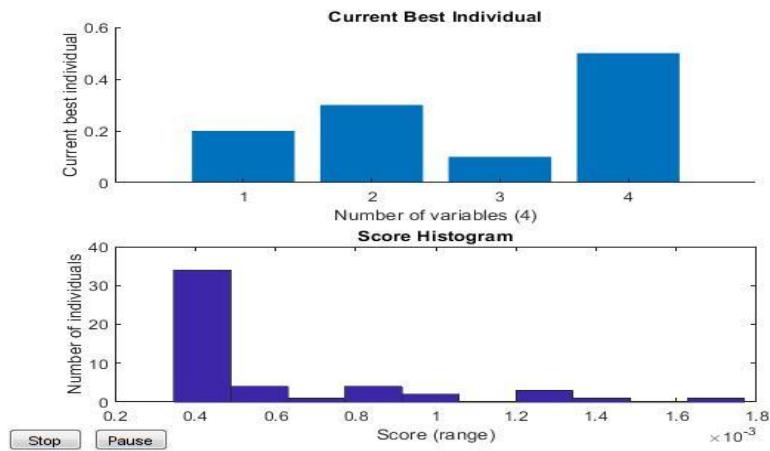


Figure 3. MATLAB Result for 11-bar Truss with 4 Damage Members (1, 5, 7, 10)

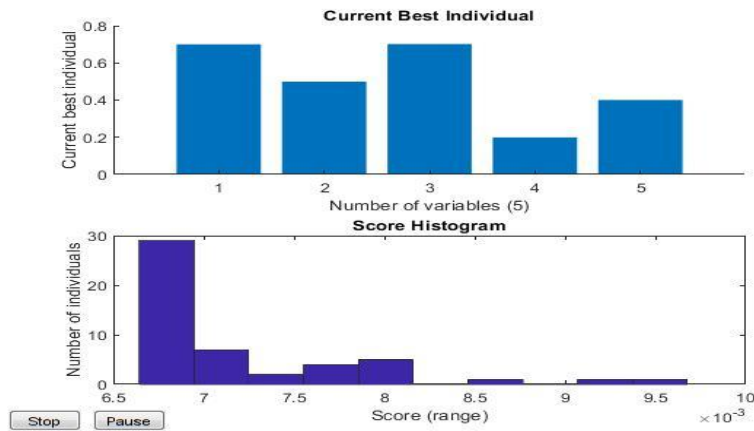


Figure 4. MATLAB Result for 11-bar Truss with 5 Damage Members (1, 3, 6, 7, 9)

Another case that has been studied is a 48-bar 3D truss, which contains 1 bay, 3 stories, and 2 frames. Each frame consists of columns, beams and X braces in each bay and stories, including the connecting bays.

Table 11. 48 Bar Truss Example with 5 Damage Elements (Case 4)

Damage Element	Damage Severity	Detected elements by existed alg.
5	90%	detected
13	80%	Not detected
20	60%	Not detected
35	90%	detected
37	20%	Not detected

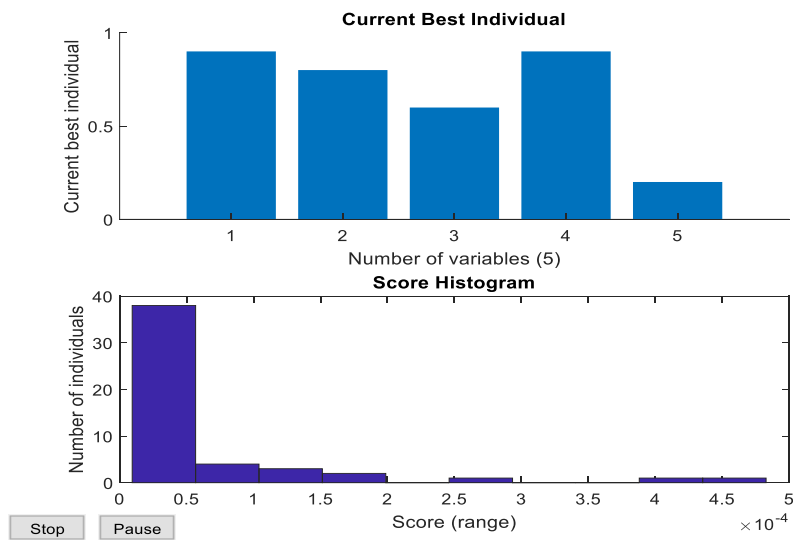


Figure 5. MATLAB Result for 48-bar Truss Damage Members (5, 13, 20, 35, 37)

Another example is a bridge with 10 bays, each 8 feet, 8 stories, each 8 feet and 6 frames, each 8 feet. Each frame consists of columns, beams, and X brace frames. This example is a simply supported has 1782 degrees of freedom, 594 nodes, and 3288 members and is a larger size problem. This structure has been used to show the time efficiency as well as accuracy of the proposed method.

In this example, 5 elements have been identified as damage by the proposed algorithm correctly. The damage severity of members is varied, which have been detected by the program correctly. Also, the computing time is reduced by using 2 processors in parallel computation. Computation time using different number of processors is reported in Table 12. The results can be found in Table 13 and Figure 6.

Table 12. Computation Time in Parallel Performance for Larger Scale Problem

Number of processors	Time (second)
1	1375.4501
2	941.3579
3	898.6813
4	867.9016

Table 13. Larger Scale Truss Example with 5 Damage Elements (Case 5)

Damage Element	Damage Severity	Detected elements by existed alg.
10	80%	detected
37	70%	Not detected
55	90%	Not detected
529	75%	Not detected
705	40%	Not detected

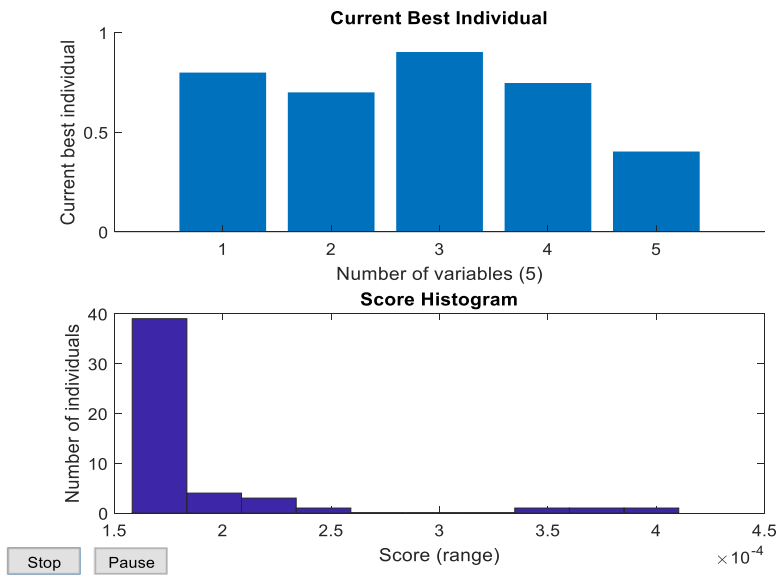


Figure 6. MATLAB Result for Larger Truss Damage Members (10, 37, 55, 529, 705)

There are some cases in which, even by considering large number of mode-shapes, existing algorithms [13, 14] will not be able to detect “all” damage elements, especially when the number of damage elements is more than 3. Using our suggested “simple rule of thumb,” however, existing algorithms [13, 14] will be able to detect “all” damage members.

Subspace iteration in combination with Jacobi rotation algorithm have been implemented into the damage detection problem for computing the few lowest eigen pairs. Combination of subspace iteration and MATLAB “eig” built-in function have also been used for performance evaluation. In almost all numerical cases considered in this study, this combined (subspace iteration and MATLAB “eig”) algorithm does not converge to the correct eigen-pairs. These mentioned numerical results have clearly shown that our proposed PSI-JT algorithm is more robust (reliable) as compared to MATLAB built-in “EIG” function.

CHAPTER 4

CONCLUSIONS

Serial Jacobi transformation algorithm for the solution of “standard eigen-problems” is re-visited to facilitate the explanation of the proposed parallel transformation algorithm, for which computational efficiency can be realized in this study through “pattern recognition” for the development of “explicit formulas” to avoid costly matrix time matrix operations.

In this work, the Jacobi transformation algorithm is embedded inside the subspace iteration algorithm to calculate the generalized eigen-problem of the monitored structure. To provide the effective computational procedure, a parallel computing strategy based on the idea of making several off-diagonal terms to be simultaneously driven to zero is used for the Jacobi transformation algorithm, which is so-called parallel subspace iteration and Jacobi transformation (PSI-JT) algorithm. The results depict the accuracy and time efficiency of the proposed algorithm.

Numerical results obtained from this study have indicated that our proposed generalized Jacobi transformation is more robust and reliable as compared to MATLAB eigen-solver. Specifically, for obtaining more eigen pairs, the PSI-JT algorithm is shown to produce more robust results.

The proposed parallel Jacobi transformation for the solution of “generalized eigen-problems” has also been incorporated into our “improved damage detection” algorithm. Computational efficiency and robust behavior for the entire proposed procedures (eigen-solution, damage detection and damage quantification) can be validated through several academic and real-life numerical examples.

For damage members severity estimation, an optimization problem needs to be solved repeatedly to converge to the correct solution. Using PSI-JT algorithm is depicted to produce robust solution in damage severity quantification.

REFERENCES

1. Carden, E. P. (2004). Vibration Based Condition Monitoring: A Review, *Structural Health Monitoring*, 3(4), 355–377
2. Wei Fan, & Pizhong Qiao. (2011). Vibration-Based Damage Identification Methods: A Review and Comparative Study, *Structural Health Monitoring*, 10(1), 83–111.
3. Das, S., Saha, P., & Patro, S. K. (2016). Vibration-Based Damage Detection Techniques Used for Health Monitoring of Structures: A Review, *Journal of Civil Structural Health Monitoring*, 6(3), 477–507.
4. K.J. Bathe, *Finite Element Procedures*, Prentice Hall Publisher (1996)
5. J.N. Reddy, *Finite Element Method*, Third Edition, McGraw-Hill Publisher (2006)
6. Gene H. Golub, Charles F. Van Loan, *Matrix Computations*, Fourth Edition (2013)
7. D.T. Nguyen, *Finite Element Methods: Parallel-Sparse Statics and Eigen-Solutions*, Springer Publisher (2006)
8. M.T. Heath, *Scientific Computing: An Introductory Survey*, McGraw-Hill Publisher (1997)
9. Ahmed H. Sameh, "On Jacobi and Jacobi-Like Algorithms for a Parallel Computer," *Mathematics of Computation*, Vol. 25, No. 115, pages 579-590 (July 1971).
10. S. Pissanetzky, *Sparse Matrix Technology*, Academic Press Publisher (1984)
11. T Vo-Duy, V Ho-Huu, H Dang-Trung, T Nguyen-Thoi, "A two-step approach for damage detection in laminated composite structures using modal strain energy method and an improved differential evolution algorithm," *Journal of Composite Structures*, Vol. 147, pages 42-53, Elsevier Publisher (2016/7/1)

12. D Dinh-Cong, T Vo-Duy, V Ho-Huu, A Tran-Viet, T Nguyen-Thoi, "An efficient multi-stage optimization approach for damage detection in plate structures," *Advances in Engineering Software*, Vol 112, pages 76-87 (2017/10/1)
13. T Vo-Duy, N Nguyen-Minh, H Dang-Trung, A Tran-Viet, T Nguyen-Thoi, "Damage assessment of laminated composite beam structures using damage locating vector (DLV) method," *Frontiers of Structural and Civil Engineering* Vol. 9 (Issue 4), pp. 457-465 (2015/12/1)
14. D Dinh-Cong, T Vo-Duy, N Nguyen-Minh, V Ho-Huu, T Nguyen-Thoi, "A two-stage assessment method using damage locating vector method and differential evolution algorithm for damage identification of cross-ply laminated composite beams," *Advances in Structural Engineering*, Vol. 20 (issue 12), pages 1807-1827 (2017/12), SAGE Publications
15. D Dinh-Cong, H Dang-Trung, T Nguyen-Thoi, "An efficient approach for optimal sensor placement and damage identification in laminated composite structures," *Advances in Engineering Software*, Vol 119, pages 48-59 (2018/5/1), Elsevier Publisher.
16. D Dinh-Cong, V Vo-Duy, T Nguyen-Thoi, "Damage assessment in truss structures with limited sensors using a two-stage method and model reduction," *Applied Soft Computing*, Vol. 66, pages 264-277 (2018/5/1), Elsevier Publisher
17. D Dinh-Cong, V Vo-Duy, V Ho-Huu, T Nguyen-Thoi, "Damage assessment in plate-like structures using a two-stage method based on modal strain energy change and Jaya algorithm', *Inverse Problems in Science and Engineering*, Vol. 27 (Issue 2), pages 166-189 (2019/2/1), Taylor & Francis Publisher

18. D Dinh-Cong, S Pham-Duy, T Nguyen-Thoi, "Damage detection of 2D frame structures using incomplete measurements by optimization procedure and model reduction," *Journal of Advanced Engineering and Computation*, Vol. 2 (Issue 3), pages 164-173 (2018/9/30),
19. Bernal, D. (2002). Load Vectors for Damage Localization, *Journal of Engineering Mechanics*, 128(1), 7–14.
20. Gao, Y., Spencer, B. F., & Bernal, D. (2007). Experimental Verification of the Flexibility-Based Damage Locating Vector Method, *Journal of Engineering Mechanics*, 133(10), 1043–1049.
21. Dinh-Cong, D., Vo-Duy, T., Nguyen-Minh, N., Ho-Huu, V., & Nguyen-Thoi, T. (2017). A Two-Stage Assessment Method Using Damage Locating Vector Method and Differential Evolution Algorithm for Damage Identification of Cross-Ply Laminated Composite Beams, *Advances in Structural Engineering*, 20(12), 1807–1827.
22. Nguyen-Thoi, T., Tran-Viet, A., Nguyen-Minh, N., Vo-Duy, T., & Ho-Huu, V. (2018). A Combination of Damage Locating Vector Method (DLV) and Differential Evolution Algorithm (DE) for Structural Damage Assessment, *Frontiers of Structural and Civil Engineering*, 12(1), 92–108.
23. <https://www.cise.ufl.edu/research/sparse/matrices/HB/>
24. <https://sparse.tamu.edu/HB>

APPENDICES

APPENDIX 1

Old Dominion University (ODU) MATLAB Source Code for “Parallel Subspace Iteration with Jacobi Transformation”

A.1 Subspace source code with Jacobi Rotation Combination

Below the MATLAB source code of subspace iteration with Jacobi rotation implementation is represented.

```
clear all

close all

clc

% Define K and M matrices

% A = [5 -4 1 0;-4 6 -4 1;1 -4 6 -4;0 1 -4 5];

% B = [2 -1 0 0;-1 4 -1 0;0 -1 4 -1;0 0 -1 2];

N = load('bcsstk13');

N_1 = N.Problem.A;

A = full(N_1);
```

```

B=eye(size(A,1));

% Input lowest eigen value desired

L =77;

% Deifine first guess

x_k = zeros(size(A,2),(4*L));

for j = 1:size(A,2)

    for i = 1:(4*L)

        x_k(i,i) = 1;

    end

end

x_k = x_k(1:size(A,2), 1:(4*L));

% Subspace code

max_Abs_error_norm = 1;

ecol = 1;

err = 1;

X_bar = x_k;

m_n=0;

% tic

while max_Abs_error_norm > 10e-7 || ecol > 10e-3

    m_n=m_n+1;

    B_mod = B*X_bar;

```



```

X_bar = A\B_mod;

A_R = X_bar'*A*X_bar;

B_R = X_bar'*B*X_bar;

[val,phi,sweep]=eigenpair_generalized_Parallel_2(A_R,B_R);

% sort

[val,ind] = sort(val);

phi = phi(:,ind);


X = X_bar*phi;

for i=1:(L)

    Abs_error_norm(i) = norm(A*X(:,i)-val(i)*B*X(:,i));

end

max_Abs_error_norm = norm(Abs_error_norm);

X_bar = X;

if m_n~=1

    for i=1:L

        ecol_1(i) = norm(val(i) - val_store(i));

        ecol = norm(ecol_1);

    end

    val_store = val;

else

end

```

```

val_store = val;

end

%%%%%% Check

[vc,vl]=eig(A,B);

sval=sort(abs(val));

for i=1:L

decc(i) = vl(i,i) - sval(i);

end

n_decc = norm(decc);

```

A.2 Subspace source code with MATLAB “EIG” Built-in function

Follow Subspace iteration source code with MATLAB EIG built in function is shown.

```

clear all

close all

clc

% Define K and M matrices

% A = [5 -4 1 0;-4 6 -4 1;1 -4 6 -4;0 1 -4 5];

% B = [2 -1 0 0;-1 4 -1 0;0 -1 4 -1;0 0 -1 2];

N = load('bcsstk13');

```

```

N_1 = N.Problem.A;

A = full(N_1);

B=eye(size(A,1));

% Input lowest eigen value desired

L =77;

% Deifine first guess

x_k = zeros(size(A,2),(4*L));

for j = 1:size(A,2)

    for i = 1:(4*L)

        x_k(i,i) = 1;

    end

end

x_k = x_k(1:size(A,2), 1:(4*L));

% Subspace code

max_Abs_error_norm = 1;

ecol = 1;

err = 1;

X_bar = x_k;

m_n=0;

% tic

while max_Abs_error_norm > 10e-7 || ecol > 10e-3

    m_n=m_n+1;

```

```

B_mod = B*X_bar;

X_bar = A\B_mod;

A_R = X_bar'*A*X_bar;

B_R = X_bar'*B*X_bar;

[phi,val]=eig(A_R,B_R);

% sort

[val,ind] = sort(abs(diag(val)));

phi = phi(:,ind);

X = X_bar*phi;

for i=1:(L)

    Abs_error_norm(i) = norm(A*X(:,i)-val(i)*B*X(:,i));

end

max_Abs_error_norm = norm(Abs_error_norm);

X_bar = X;

if m_n~=1

    for i=1:L

        ecol_1(i) = norm(val(i) - val_store(i));

        ecol = norm(ecol_1);

    end

    val_store = val;

else

end

```

```

val_store = val;

end

%%%%%% Check

[vc,vl]=eig(A,B);

sval=sort(abs(val));

for i=1:L

decc(i) = vl(i,i) - sval(i);

end

n_decc = norm(decc);

```

A.3 Jacobi Rotation Source Code

Jacobi rotation source code using the explicit formula described in the previous chapters is presented.

```

function[val,phi,sweep]=eigenpair_generalized_Parallel_2(k,M)

n=size(k,2);

m = (n+1)/2;

m = fix(m);

nprocessor = n/2;

phi = eye(size(k,1));

```

```

nn=1;

sweep=0;

while nn~=0

    nn=0;

    sweep=sweep+1;

    for rr=1:size(k,1)-1

        if rr <= m-1

            for i = 1:nprocessor

                q(i) = m - rr +i;

                if q(i)<= (2*m - 2*rr) && q(i)>=(m-rr+1)

                    p(i) = (2*m - 2*rr +1)-q(i);

                elseif q(i)<= (2*m -rr-1) && q(i)>(2*m-2*rr)

                    p(i) = (4*m - 2*rr) - q(i);

                elseif q(i)> (2*m-rr-1)

                    p(i) = n;

                end

                if q(i)<p(i)

                    pc=p(i);

                    p(i)=q(i);

                    q(i)=pc;

                end

            end

        end

    end

```

```

elseif rr >= m

    for i = 1:nprocessor

        %    if k==(2*m-1) && i==2

        %        q(i) = 3*m - k -1;

        %    else

            q(i) = 4*m - n - rr +i-1;

        %    end

        if q(i)> (4*m - 2*rr - 1)

            p(i) = (6*m - 2*rr -1)-q(i);

        elseif q(i)>=(2*m -rr+1) && q(i)<=(4*m-2*rr-1)

            p(i) = (4*m - 2*rr) - q(i);

        elseif q(i)< (2*m-rr+1)

            p(i) = n;

        end

        if q(i)<p(i)

            pc=p(i);

            p(i)=q(i);

            q(i)=pc;

        end

    end

end

end

```

```

% p1=zeros(size(k));

% for i=1:size(k,1)

%   p1(i,i)=1;

% end

p1=eye(size(k));

kbar = zeros(size(k,1));

k_bar = zeros(1,size(p,2));

x = zeros(1,size(p,2));

lambda = zeros(1,size(p,2));

alpha = zeros(1,size(p,2));

for i = 1:size(p,2)

    if (k(p(i),p(i))/M(p(i),p(i)))==(k(q(i),q(i))/M(q(i),q(i)))==(k(p(i),q(i))/M(p(i),q(i)))

        alpha(i) = 0;

        lambda(i) = (-1)*(k(p(i),q(i))/k(q(i),q(i)));

    else

kbar(p(i),p(i)) = k(p(i),p(i))*M(p(i),q(i))-M(p(i),p(i))*k(p(i),q(i));

kbar(q(i),q(i)) = k(q(i),q(i))*M(p(i),q(i))-M(q(i),q(i))*k(p(i),q(i));

k_bar(i) = k(p(i),p(i))*M(q(i),q(i))-k(q(i),q(i))*M(p(i),p(i));

if k_bar(i)>=0

    x(i) = (k_bar(i)/2)+sqrt((k_bar(i)/2)^2+kbar(p(i),p(i))*kbar(q(i),q(i)));

elseif k_bar(i)<0

    x(i) = (k_bar(i)/2)-sqrt((k_bar(i)/2)^2+kbar(p(i),p(i))*kbar(q(i),q(i)));

```


end

lambda(i) = (-1)*(kbar(p(i),p(i))/x(i));

alpha(i) = kbar(q(i),q(i))/x(i);

end

p1(p(i),q(i))=alpha(i);

p1(q(i),p(i))=lambda(i);

end

phi=phi*p1;

%%%

%Creat new k based on my formula

parfor pi=1:nprocessor

Tempo1 = zeros(p(pi) ,1);

Tempo1_M = zeros(p(pi) ,1);

Tempo2 = zeros(q(pi) ,1);

Tempo2_M = zeros(q(pi) ,1);

pSubs = zeros(p(pi) ,2); %new

qSubs = zeros(q(pi) ,2);

for irow = 1:p(pi)

[xx,inside_angle] = find(irow==[p;q]);

Tempo1(irow) = [(xx-1)*alpha(inside_angle)+(2-xx)]* ...

```

[k(p(inside_angle),p(pi))+lambda(pi)*k(p(inside_angle),q(pi))] + ...

[(2-xx)*lambda(inside_angle)+(xx-1)]* ...

[k(q(inside_angle),p(pi))+lambda(pi)*k(q(inside_angle),q(pi))];

Tempo1_M(irow) = [(xx-1)*alpha(inside_angle)+(2-xx)]* ...

[M(p(inside_angle),p(pi))+lambda(pi)*M(p(inside_angle),q(pi))] + ...

[(2-xx)*lambda(inside_angle)+(xx-1)]* ...

[M(q(inside_angle),p(pi))+lambda(pi)*M(q(inside_angle),q(pi))];

pSubs(irow,:)= [irow,p(pi)]; %new

end

for irow = 1:q(pi)

[xx,inside_angle] = find(irow==[p;q]);

Tempo2(irow) = [(xx-1)*alpha(inside_angle)+(2-xx)]* ...

[alpha(pi)*k(p(inside_angle),p(pi))+k(p(inside_angle),q(pi))] + ...

[(2-xx)*lambda(inside_angle)+(xx-1)]* ...

[alpha(pi)*k(q(inside_angle),p(pi))+k(q(inside_angle),q(pi))];

Tempo2_M(irow) = [(xx-1)*alpha(inside_angle)+(2-xx)]* ...

[alpha(pi)*M(p(inside_angle),p(pi))+M(p(inside_angle),q(pi))] + ...

[(2-xx)*lambda(inside_angle)+(xx-1)]* ...

[alpha(pi)*M(q(inside_angle),p(pi))+M(q(inside_angle),q(pi))];

qSubs(irow,:)= [irow,q(pi)]; %new

end

```

```

subsCell{pi,1}=[pSubs;qSubs]; %new

kValCell{pi,1}=[Tempo1;Tempo2];

MValCell{pi,1}=[Tempo1_M;Tempo2_M];


%Assign tempos to k

%   for irow = 1:p(pi)

%       k_1(irow,p(pi)) = Tempo1(irow);

%       k_1(p(pi),irow) = Tempo1(irow);

%       M_1(irow,p(pi)) = Tempo1_M(irow);

%       M_1(p(pi),irow) = Tempo1_M(irow);

%   end

%   for irow = 1:q(pi)

%       k_1(irow,q(pi)) = Tempo2(irow);

%       k_1(q(pi),irow) = Tempo2(irow);

%       M_1(irow,q(pi)) = Tempo2_M(irow);

%       M_1(q(pi),irow) = Tempo2_M(irow);

%   end

end

subs=cell2mat(subsCell);

kVal=cell2mat(kValCell);

MVal=cell2mat(MValCell);

```

```

k_1=accumarray(subs,kVal,size(k));

M_1=accumarray(subs,MVal,size(M));

k=k_1 + tril(k_1.',-1); %make symmetric

M=M_1 + tril(M_1.',-1);

end

for ki=1:size(k,1)

    sum=0;

    if k(ki,ki)~=0

        for kj=1:size(k,1)

            if kj==ki

                kj=kj+1;

            else

                sum = sum + abs(k(ki,kj));

            end

        end

    end

    if abs(k(ki,ki))>(100*sum)

        nn=nn+0;

    else

        nn=nn+1;

```

```
    end  
    end  
end  
end  
  
for i = 1:size(k,1)  
    val(i)=k(i,i)/M(i,i);  
end  
  
end
```

APPENDIX 2

One of the examples is a 2003x2003 matrix (a Symmetrical Stiffness Matrix, which represents the Fluid Flow Generalized Eigen-Problems), is also included. If the number of requested eigen-pairs is 63, then MATLAB built-in function (EIG) will not be able to converge to the correct solution.

However, if we replace MATLAB built-in function (EIG) with our Generalized Subspace Iteration with Jacobi Rotation source code, then correct eigen-solutions have been obtained.

The input file has been downloaded from Texas A&M website, and also have been adopted and published in other valid websites described in the related references [23, 24]. Following are the complete information and figures of matrices selected from these sources [23, 24] and used in this dissertation work.

B.1 Symmetrix stiffness matrix, module of an offshore platform

This example is a real-world symmetric stiffness matrix, shows module of an offshore platform. The figure is shown in Figure 7. Matrix properties consist of number of rows and columns, number of nonzero terms and other related features, are represented in the Table 14.

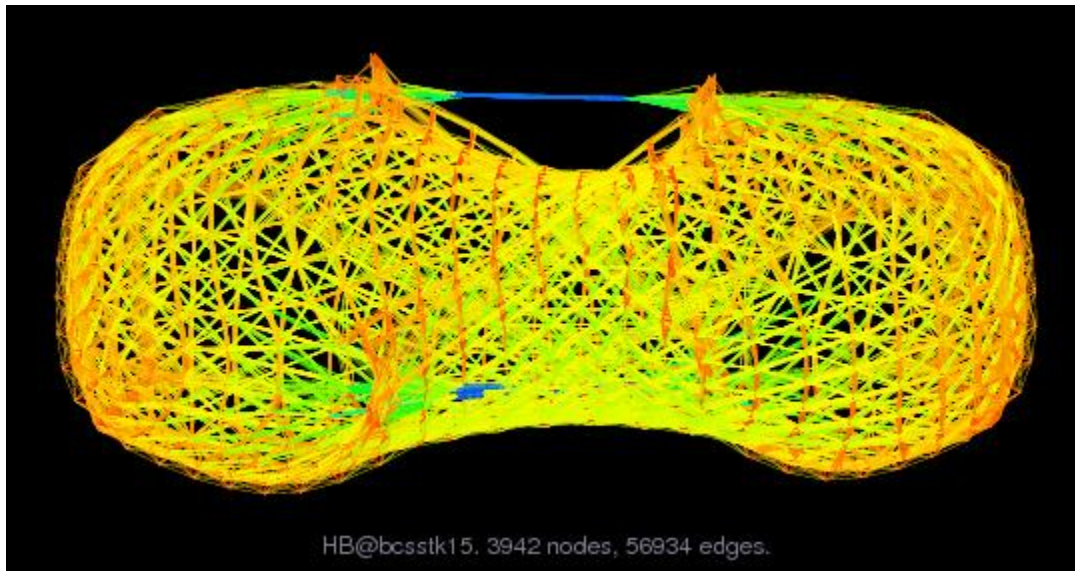


Figure 7. Symmetric Stiffness Matrix, Module of an Offshore Platform

Table 14. Symmetric Stiffness Matrix, Module of an Offshore Platform, Properties

Matrix Properties	
number of rows	3,948
number of columns	3,948
nonzeros	117,816
structural full rank?	yes
structural rank	3,948
explicit zero entries	0
nonzero pattern symmetry	symmetric
numeric value symmetry	symmetric
type	real
structure	symmetric
Cholesky candidate?	yes
positive definite?	yes

This matrix is authored by M. Will, and is edited by I. Duff, R. Grimes, J. Lewis [23]. This matrix is a full matrix, and as it is shown in Figure 7, the matrix is related to the 3D problem.

B.2 Symmetric Stiffness Matrix, Fluid Flow Generalized Eigenvalues

This example is also a real-world symmetric stiffness matrix, extracted from fluid Flow Generalized Eigenvalues problem. The figure of the matrix is represented in Figure 8, and the matrix properties are described in Table 15.

This matrix is authored by J. Lewis, and is edited by I. Duff, R. Grimes, J. Lewis [23]. This matrix is a computational fluid dynamic 3D problem.

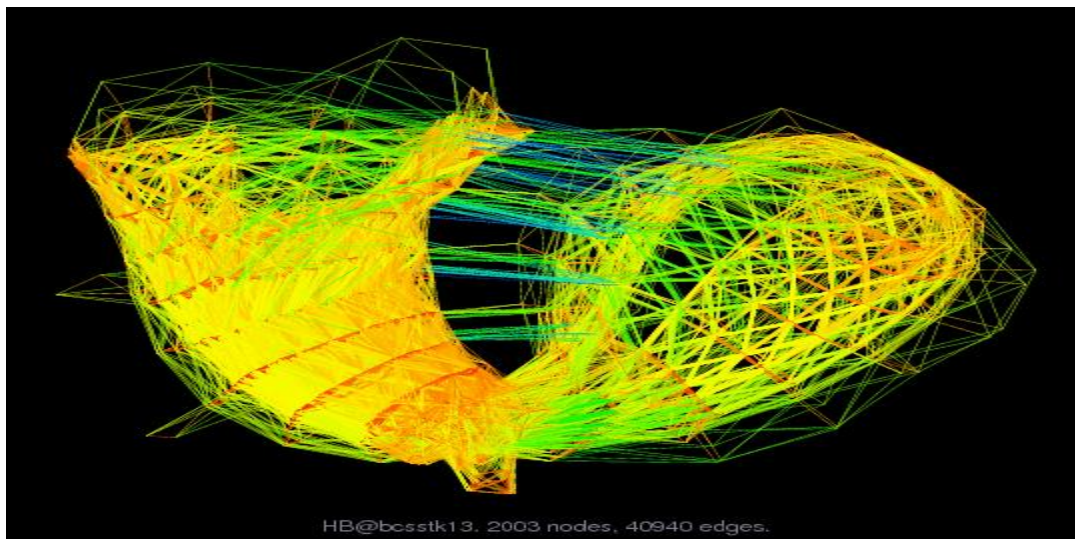


Figure 8. Symmetric Stiffness Matrix, Fluid Flow Generalized Eigenvalues

Table 15. Symmetric Stiffness Matrix, Fluid Flow Generalized Eigenvalues, Properties

Matrix Properties	
umber of rows	2,003
number of columns	2,003
nonzeros	83,883
structural full rank?	yes
structural rank	2,003
explicit zero entries	0
nonzero pattern symmetry	symmetric
numeric value symmetry	symmetric
type	real
structure	symmetric
Cholesky candidate?	yes
positive definite?	yes

B.3 Symmetric Stiffness Matrix, Buckling of Hot Washer

Another real-world problem is presented in this section. The data is extracted from the websites mentioned in the previous sections [23, 24]. Table 16 shows the properties of this matrix, and Figure 9 demonstrates the figure of the matrix.

This matrix is authored by J. Lewis, and is edited by I. Duff, R. Grimes, J. Lewis [23]. As it is clear from the name of the name, this is matrix is extracted from a structural 3D problem.

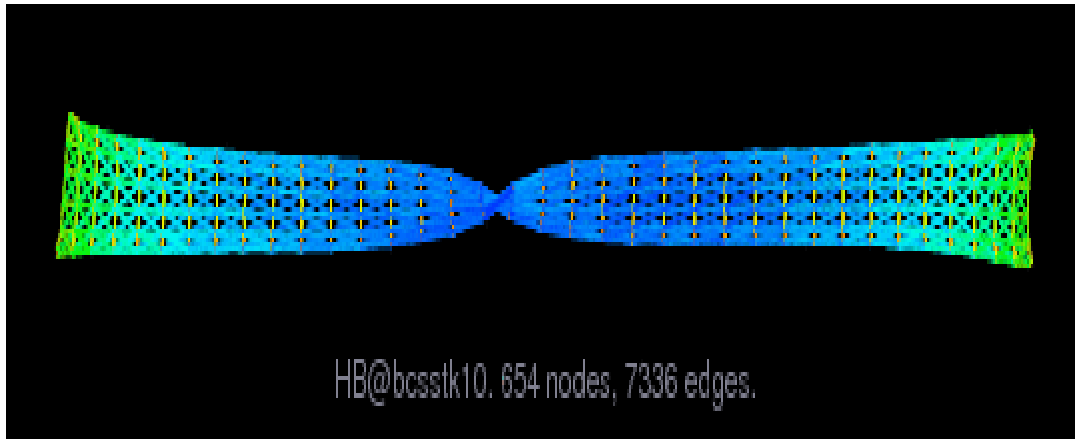


Figure 9. Symmetric Stiffness Matrix, Buckling of Hot Washer

B.4 Symmetric Stiffness Matrix, Medium Test Problem, Lumped Masses

This is also another structural 3D problem with lower number of rows and columns compare to the previous cases. This matrix is authored by J. Lewis, and is edited by I. Duff, R. Grimes, J. Lewis [23].

More information about matrix properties is described in Table 17, and the figure of the matrix is shown in Figure 10.

Table 16. Symmetric Stiffness Matrix, Buckling of Hot Washer, Properties

Matrix Properties	
number of rows	1,086
number of columns	1,086
nonzeros	22,070
structural full rank?	yes
structural rank	1,086
explicit zero entries	0
nonzero pattern symmetry	symmetric
numeric value symmetry	symmetric
type	real
structure	symmetric
Cholesky candidate?	yes
positive definite?	yes

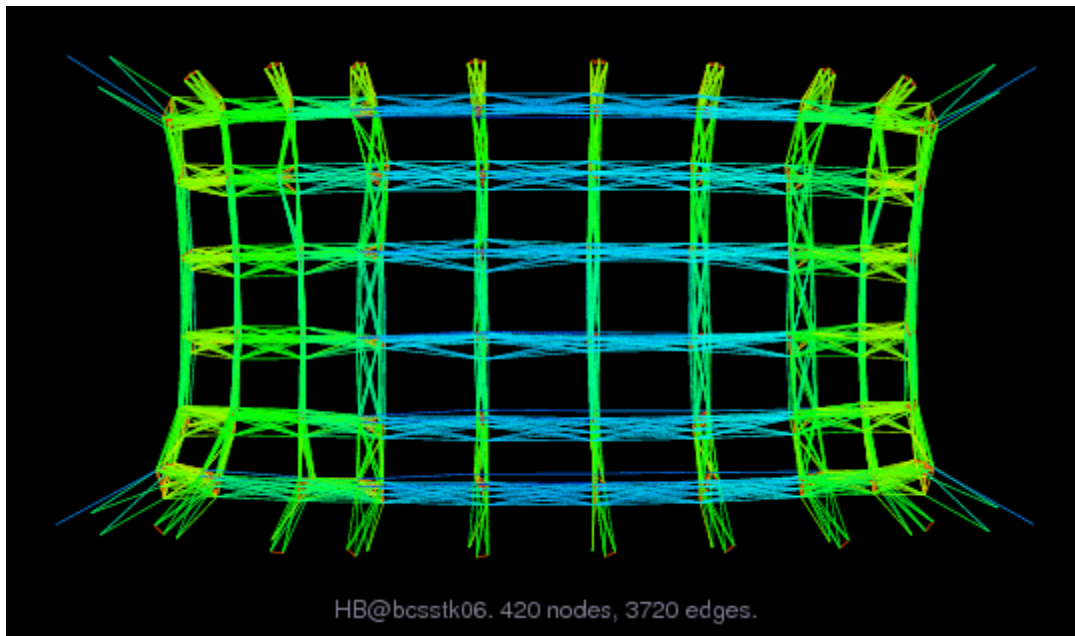


Figure 10. Symmetric Stiffness Matrix, Medium Test Problem, Lumped Masses

Table 17. Symmetric Stiffness Matrix, Medium Test Problem, Lumped Masses, Properties

Matrix Properties	
number of rows	420
number of columns	420
nonzeros	7,860
structural full rank?	yes
structural rank	420
explicit zero entries	0
nonzero pattern symmetry	symmetric
numeric value symmetry	symmetric
type	real
structure	symmetric
Cholesky candidate?	yes
positive definite?	yes

B.5 Symmetric Stiffness Matrix, Transformation Tower, Lumped Masses

This symmetric stiffness matrix is related to a 3D structural problem. It is authored by J. Lewis, and is edited by I. Duff, R. Grimes, J. Lewis [23]. It is worth mentioning that this matrix is one of the small size matrices that has been used in this research for authorizing PSI-JT algorithm.

The figure of this matrix is shown in Figure 11. The properties of the matrix is described in detail in Table 18.

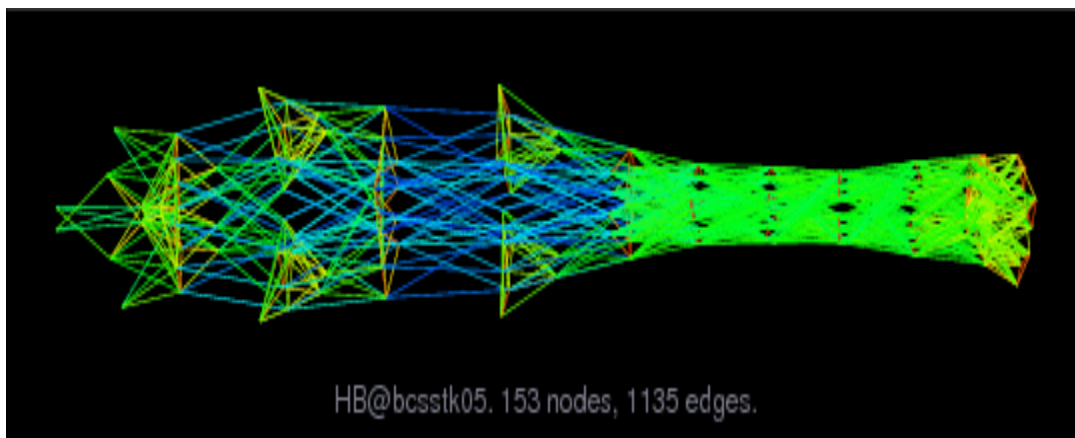


Figure 11. Symmetric Stiffness Matrix, Transformation Tower, Lumped Masses

Table 18. Symmetric Stiffness Matrix, Transformation Tower, Lumped Masses, Properties

Matrix Properties	
number of rows	153
number of columns	153
nonzeros	2,423
structural full rank?	yes
structural rank	153
explicit zero entries	0
nonzero pattern symmetry	symmetric
numeric value symmetry	symmetric
type	real
structure	symmetric
Cholesky candidate?	yes
positive definite?	yes

In this research all of the previous cases described in detail in this section, is used to test and validate the performance of PSI-JT algorithm. Looking at the figures of these cases, it is clear that they are completely different in the formation, and are not have a lot in common, but being sparse symmetric positive definite.

APPENDIX 3

Since the code for this section is so large and also Jacobi rotation source code has been presented in Appendix A.3, the complete code for this part will not be mentioned here. However, the Source code for truss generation that has been used to create any size 2D and 3D trusses is presented in this section.

C.1 Truss Creation Source Code

The following source code was written by the dissertation author in MATLAB and is able to create 2D and 3D truss. In this code the user needs to specify very short input data, such as number of bays, number of storied, 2D or 3D format, etc., and the code is able to create the truss and all the features, such as connectivity table, etc., by itself.

```
close all
clear all
clc

% User Inputs
fprintf('\n');
nbays=input('ENTER THE NUMBER OF Bays:-');
fprintf('\n');
```

```

fprintf('*****
*****\n');

% if nbays>=1

fprintf('\n');

Length=input('ENTER THE Length of Each Bay:-');

fprintf('\n');

% end

fprintf('*****
*****\n');

fprintf('\n');

nstories=input('ENTER THE NUMBER OF Stories:-');

fprintf('\n');

fprintf('*****
*****\n');

fprintf('\n');

Height=input('ENTER THE Height OF Each Stories:-');

fprintf('\n');

fprintf('*****
*****\n');

fprintf('\n');

nframes=input('ENTER THE NUMBER OF Frames:-');

fprintf('\n');

fprintf('*****
*****\n');

fprintf('\n');

Width=input('ENTER THE Width OF Two Consecutive Frames:-');

fprintf('\n');

fprintf('*****
*****\n');

```



```

fprintf('\n');
n_node_element=input('ENTER THE NUMBER OF Nodes per Element:-');
fprintf('\n');
fprintf('*****
*****\n');
fprintf('\n');
% Truss Dimension
global num_dof_node
global num_dof_ele
num_dof_node=input('ENTER THE NUMBER OF SPATIAL DIMENSIONS:-');
num_dof_ele=n_node_element*num_dof_node;
fprintf('\n');
fprintf('*****
*****\n');
fprintf('\n');
damage_ele=input('ENTER THE damage element and severity [ele sve;..]:-');
fprintf('\n');
fprintf('*****
*****\n');
fprintf('\n');
a_ver=input('ENTER THE Area of Vertical Area:-');
fprintf('\n');
fprintf('*****
*****\n');
fprintf('\n');
a_hor=input('ENTER THE Area of Horizontal Area:-');
fprintf('\n');
fprintf('*****
*****\n');

```

```

fprintf('\n');
a_diag=input('ENTER THE Area of Diagonal; Area:-');
fprintf('\n');

tic

% Compute Number of Nodes
global num_nod
num_nod = (nstories+1)*(nbays+1)*(nframes);
num_nod_fram = (nstories+1)*(nbays+1);

% nodes coordinates
global nod_coor
nod_coor = zeros(num_nod,num_dof_node);
e = 1;
for i = 1:nframes
    for k = 1:nstories+1
        for j = 1:nbays+1
            nod_coor(e,:) = [0+(j-1)*Length, 0+(k-1)*Height, 0+(i-1)*Width];
            e = e+1;
            if e == num_nod+1
                break
            end
        end
    end
end
end
end

```

```
% Number of Vertical Elements
```

```
num_ver_ele = nstories*(nbays+1)*nframes;
```

```
num_ver_frame = nstories*(nbays+1);
```

```
% Number of Horizontal Elements
```

```
num_hor_ele = nstories*nbays*nframes + nstories*(nbays+1)*(nframes-1);
```

```
% Number of Diagonal Elements
```

```
num_diag_ele = 2*nstories*(2*nbays*nframes+nframes-nbays-1);
```

```
%Total Number of Elements
```

```
global num_ele
```

```
num_ele = num_ver_ele + num_hor_ele + num_diag_ele;
```

```
% Construct the Connectivity Matrix
```

```
global ele_nod
```

```
global A
```

```
ele_nod = zeros(num_ele,n_node_element);
```

```
% Vertical Elements Connectivity
```

```
for j=1:nframes
```

```
    for i=(1+num_ver_frame*(j-1)):(num_ver_frame*j)
```

```
        ele_nod(i,:) = [i+(num_nod_fram-num_ver_frame)*(j-1), ...
```

```
            i+(nbays+1)+(num_nod_fram-num_ver_frame)*(j-1)];
```

```
        A(i) = a_ver;
```

```
    end
```

```
end
```

```
%Horizontal Elements Connectivity Matrix
```

```

[x,y] = find(nod_coor(:,2)~=0);
i = i + 1;
for e = 1:nframes
    for k = 1:nstories
        for j = 1:nbays
            ele_nod(i,:) = [x(j+(k-1)*(nbays+1)+(e-1)*(num_nod_fram-(nbays+1))), ...
                x(j+1+(k-1)*(nbays+1)+(e-1)*(num_nod_fram-(nbays+1)))];
            A(i) = a_hor;
            i = i + 1;
        end
    end
end

```

```

if nframes>1
    for j = 1:((num_nod_fram-(nbays+1))*(nframes-1))
        ele_nod(i,:) = [x(j),x(j)+num_nod_fram];
        A(i) = a_hor;
        i = i+1;
    end
end

```

%Diagonal Elements Connectivity Matrix

```

i = i - 1;
for j=1:nbays
    [x1,y1] = find(nod_coor(:,1)==(Length*(j-1)));
    [x2,y2] = find(nod_coor(:,1)==(Length*j));
    sx1 = size(x1,1);
    for e = 1:nframes

```

```

for k = 1:((sx1/nframes)-1)
    i = i+1;
    ele_nod(i,:) = [x1(k+((e-1)*(nstories+1))),x2(k+1+((e-1)*(nstories+1)))];
    A(i) = a_diag;
    i = i+1;
    ele_nod(i,:) = [x2(k+((e-1)*(nstories+1))),x1(k+1+((e-1)*(nstories+1)))];
    A(i) = a_diag;
end
end
end

if nframes>1
    for j=1:nbays+1
        [x3,y3] = find(nod_coor(:,1)==(Length*(j-1)));
        sx1 = size(x3,1);
        for e=1:(nframes-1)
            for k = 1:((sx1/nframes)-1)
                i = i+1;
                ele_nod(i,:) = [x3(k+(e-1)*(nstories+1)),x3(k+(e-1)*(nstories+1)+(nstories+2))];
                A(i) = a_diag;
                i = i+1;
                ele_nod(i,:) = [x3(k+(e-1)*(nstories+1)+(nstories+1)), ...
                    x3(k+1+(e-1)*(nstories+1))];
                A(i) = a_diag;
            end
        end
    end
end
end
end

```

```

% elements degree of freedom (DOF)
global ele_dof
ele_dof = zeros(num_ele,num_dof_ele);
for j=1:num_ele
    ele_dof(j,:)=[((3*ele_nod(j,1))-2),((3*ele_nod(j,1))-1),(3*ele_nod(j,1)), ...
        ((3*ele_nod(j,2))-2),((3*ele_nod(j,2))-1),(3*ele_nod(j,2)))];
end

%Form Modulus of Elasticity and mass density
global E
for i = 1:num_ele
    E(i) = 30000;
end
global rho
rho = 9.8759999999999994e-3;

fprintf('*****\n');
fprintf('\n');
number_of_loads =input('ENTER THE Number of Loads; Number:-');
fprintf('\n');
fprintf('*****\n');
fprintf('\n');
force = zeros(num_dof_node*num_nod,1);
for j=1:number_of_loads
    node_load_app=input('ENTER THE Node Number that this Load Apply to; Node:-');

```

```

    dof_load_app=input('ENTER THE DOF of the Node that This Load Apply to; DOF:-');
    load_value=input('ENTER THE Value of the Applying Load; Value:-');
    force((3*node_load_app)-(3-dof_load_app))=load_value;
end
fprintf('\n');

%Construct Boundary Condition Vector
displacement=zeros(num_dof_node*num_nod,1);
[x4,y4] = find(nod_coor(:,2)==0);
sx4 = size(x4,1);
global BC
for j=1:sx4
    for k=1:num_dof_node
        BC(k+(3*(j-1)), 1) = (3*x4(j))-(3-k);
    end
end
end

```

APPENDIX 4

The input for damage detection & quantification problems comes from both manually (for some examples to check the accuracy of the MATLAB code), and a self-written MATLAB code to generate data such as node coordinates, element nodes, connectivity table of the input truss.

As an example, for 11-bar truss, the input data file that user needs to enter to the computer screen, for using the automatically generated data for truss, is as follows:

- Total number of nodes (6 for this example)
- Number of nodes per element (for this example 2)
- Number of degrees of freedom per node (2 for the example)
- Number of spatial dimension (2 for this example, because it is a 2D truss)
- Number of bays (2 for this example)
- Number of stories (1 for this example)
- Number of frames (0 for this example, since it is a 2D structure)
- Area of each element will be asked and should be input by the user with an enter after inputting each. (For this example: 14, 1, 11, 7, 1, 1, 6, 3, 14, 1, 1)
- Modulus of elasticity (30000 for this example)
- Members' density (for this example 0.009876)

- Number of applied loads (2 for this example)
- Degree of freedom and magnitude of the applied load ([2,10000] / enter/ [6,10000])

The outputs of sample problems for damage detection & quantification have already described and presented in earlier sections of this dissertation.

The following input is the case when the data is manually inputted. The related input information is as below:

```
%number of nodes
num_nod=6;
num_dof_node = 3;

% nodes coordinates
nod_coor=[720 0 0;720 360 0;360 0 0;360 360 0;0 0 0;0 360 0];

% connectivity table
ele_nod=[6 4;4 2;5 3;3 1;3 4;1 2;6 3;5 4;4 1;3 2;5 6];

%number of elements
num_ele=size(ele_nod,1);

% elements degree of freedom (DOF)
ele_dof=[16 17 18 10 11 12;10 11 12 4 5 6;13 14 15 7 8 9;7 8 9 1 2 3; ...
        7 8 9 10 11 12; 1 2 3 4 5 6;16 17 18 7 8 9;13 14 15 10 11 12; ...
        10 11 12 1 2 3;7 8 9 4 5 6;13 14 15 16 17 18];
num_dof_ele = 6;

% A, E, L are cross sectional area, Young's modulus, length of elements,respectively.

A(1)=14;
A(2)=1;
A(3)=11;
A(4)=7;
A(5)=1;
```

```

A(6)=1;
A(7)=6;
A(8)=3;
A(9)=14;
A(10)=1;
A(11)=1;

% E(1)=30000;
for i = 1:num_ele
    E(i) = 30000;
end

rho = 9.8759999999999994e-3;

BC = [1;2;3;4;5;6;25;26;27;28;29;30];

%Define damaged elements and their related severities
damage_ele = [1 0.8;5 0.7;10 0.9];

```

It is worth mentioning that in the “damage_ele” matrix, mentioned above, the first column shows the damage element number, and the second column shows the damage severity of the related member.

This code is written in MATLAB software. In this case the input properties, such as number of nodes, number of degrees of freedom, etc, are imported by hand for comparison reasons. However, in the bigger size problems, 48-bar truss and 594-bar truss the properties are developed by the “Truss-Creation” Source code, described in the previous sections.

VITA

Maryam Ehsaei

Department of Civil and Environmental Engineering

Old Dominion University

Norfolk VA, 23529

Maryam Ehsaei was born in Shiraz, Iran, on November 1988. After finishing her B.S. in Civil and Environmental Engineering from Fasa University, Iran, on September 2011, she received her M.S. in Structural Engineering from Shiraz University, Iran, on September 2013.