A Structured Method for the real Quadratic Eigenvalue Problem for specific Gyroscopic Systems

BY

Wade D. Rush

Submitted to the graduate degree program in Mathematics
and the Graduate Faculty of the University of Kansas
in partial fulfillment of the requirements for the degree of
Masters of Arts

Chairperson: Dr. Hongguo Xu, Math Dept.

Dr. Erik Van Vleck, Math Dept.

Dr. Weizhang Huang, Math Dept.

Date defended: 13 *November* 2008

The Thesis Committee for Wade D. Rush certifies
that this is the approved Version of the following thesis:

A STRUCTURED METHOD FOR THE REAL QUADRATIC EIGENVALUE PROBLEM FOR SPECIFIC GYROSCOPIC SYSTEMS

Committee:

Chairperson: Dr. Hongguo Xu, Math Dept.

Dr. Erik Van Vleck, Math Dept.

Dr. Weizhang Huang, Math Dept.

Date approved:

ABSTRACT

This study examines a specific numerical approach that computes the eigenvalues (normal modes) of a Quadratic Eigenvalue Problem (QEP) of the form $\left(\lambda^2\mathbf{I} + \lambda\mathbf{B} + \mathbf{C}\right)\mathbf{x} = 0$ where $\mathbf{B}$ is constrained to a real skew-symmetric matrix and $\mathbf{C}$ is constrained to a real symmetric positive definite matrix. A widely used linearization of this QEP is the companion matrix $\mathbf{A} = \begin{pmatrix} -\mathbf{B} & -\mathbf{C} \\ \mathbf{I} & \mathbf{O} \end{pmatrix}$. The goal is to find an algorithm method which diagonalizes matrix $\mathbf{A}$ without contaminating the (2,2) zero block. Once this algorithm is developed, the study measures the eigenvalue error bounds and compare its efficiency to the standard symmetric QR workhorse. Also, this approach preserves the structure of the error matrix in the same form as the QEP. In ensuring that the error matrix structure is a QEP, this algorithm provides fertile ground for future analysis in sensitivity and perturbation errors in the algorithm's eigenvalues. This study concludes that the algorithm appears to have a reasonable error bound; and it is more cost efficient in finding the eigenvalues then the symmetric QR algorithm.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem

This study examines a specific numerical approach that computes the eigenvalues (normal modes) of a Quadratic Eigenvalue Problem (QEP) of the form

$$\left(\lambda^2\mathbf{I} + \lambda\mathbf{B} + \mathbf{C}\right)\mathbf{x} = 0. \tag{1.1}$$

where $\mathbf{B}$ is constrained to a real skew-symmetric matrix and $\mathbf{C}$ is constrained to a real symmetric positive definite matrix[6]. QEPs in the form of (1.1) without the constraints have a variety of applications[6]. However, with $\mathbf{B}$ and $\mathbf{C}$ constrained to a skew-symmetric and real symmetric positive definite matrices respectively, the QEP represents a special type of the 2nd-order Ordinary Differential Equation(ODE) called the gyroscopic system in the form

$$\ddot{\mathbf{x}} + \mathbf{B}\dot{\mathbf{x}} + \mathbf{C}\mathbf{x} = 0. \tag{1.2}$$

When the Laplace transform is applied to (1.2), the 2nd-order ODE is then represented with the form of (1.1).

1.2 Motivation

In the set of all possible QEPs, there is a certain class of QEPs which model oscillations in a given system. Through the use of these QEP models, these models provide critical insights into the effects these oscillations have on the system.

In the mid sixties, NASA designed an unprecedentedly large first stage booster rocket called the Saturn 5. In its inaugural launch, the Saturn 5 immediately displayed a dangerous Pogoing effect in its structure. Pogoing is a term to describe a rocket that was exhibiting large internal frame oscillation in the midst of its ascent launch. In the Saturn 5's situation, its structural frame's resonance frequency was in near match with the rocket's pipe fuel flow frequency and the combustion chamber's pressure buildup frequency. As a result, Saturn 5's frame began to oscillate, also known as structural vibration, to a level that nearly tore the booster apart[1]. Understanding and then rectifying this unwarranted oscillation was obviously a very critical path in the success of the Apollo program.

For more contemporary QEPs applied closer to earth, traffic noise has its roots embedded in the tires' natural frequency. Much traffic noise comes from the oscillatory humming of tires. The noise has become a serious contributor to noise pollution. It seemed that at around 40km/hr, cars would create unpleasant tire noise. The problem was to eliminate the tire's internal oscillation frequencies occurring between 500Hz and 2000Hz which is the range that the human ear can detect[2]. Solving the tire noise problem would be a significant net reduction of noise in our crowded urban areas.

Finally, one of the more recent and notorious examples of QEP is the Millennium bridge located in London. The Millennium foot bridge was built across the Thames River and made its debut in June 2000[6]. However, after just two days, the engineers had to shut down the foot bridge from the public due to unstable and

dangerous oscillations. Apparently, the frequency of several hundred or around a thousand people crossing the bridge at any one time matched closely to the natural resonance frequency of the foot bridge's structure. The mass of people's aggregate footfalls on the bridge reinforce the unwanted oscillations. Preferably, the design of the bridge should have had a resonance frequency well away from any natural frequencies such as people walking or wind blowing along the river.

The millennium bridge and unaccountably many more similar problems are clear motivators for investigating QEPs which model these types of oscillatory systems. In the case of this study, the study will analyze QEPs which specifically model gyroscopic problems.

1.3 Goals

When the QEP has a skew-symmetric $\mathbf{B}$ matrix and a symmetric positive definite $\mathbf{C}$ matrix, all the eigenvalues are pure imaginary[6]. The corresponding gyroscopic system to these constraints is then stable. A common numerical approach in computing eigenvalues is to transform the QEP into a standard matrix eigenvalue problem through a linearization process. The linearized or linearization matrices have the same eigenvalues as the original QEP. Once such a linearization is obtained, standard numerical eigenvalue techniques can then be used to solve the eigenvalues of the linearization as well as those of the QEP. A widely used linearization (1.1) is the companion matrix

$$\mathbf{A} = \begin{pmatrix} -\mathbf{B} & -\mathbf{C} \\ \mathbf{I} & \mathbf{O} \end{pmatrix} \tag{1.3}$$

Because of the constraints applied, $\mathbf{C}$ is a real symmetric positive definite matrix which makes it Cholesky factorizable. Using a Cholesky-like factorization, $\mathbf{C}$ is

factorized into $\mathbf{C} = \mathbf{R}\mathbf{R}^{\mathbf{T}}$ where matrix $\mathbf{R}$ is an upper-triangular matrix. As the study will later illustrate, matrix $\mathbf{A}$ is similar to the skew-symmetric matrix

$$\mathbf{H} = \begin{pmatrix} -\mathbf{B} & -\mathbf{R} \\ \mathbf{R}^{\mathbf{T}} & \mathbf{O} \end{pmatrix} \tag{1.4}$$

To determine the eigenvalues of $\mathbf{H}$, a QR algorithm, modified from the symmetric to a skew-symmetric case, could be applied to the skew-symmetric matrix $\mathbf{H}$. However, such a QR algorithm ignores the (2,2) zero matrix block in matrix $\mathbf{H}$ and treats $\mathbf{H}$ as a general real skew-symmetric matrix. QR's failure to acknowledge the zero block has the potential for creating two significant numerical problems. Due to rounding error, the computed eigenvalues can only be considered as the exact eigenvalues to a slightly perturbed skew-symmetric matrix. Since in the perturbed matrix, the (2,2) block is usually nonzero, this perturbed skew-symmetric matrix generally does not represent a matrix derived from a quadratic eigenvalue problem of the form of (1.1). Additionally, by ignoring the (2,2) zero block, the computational costs may increase. Due to these reasons, this study has developed a new approach which recognizes and preserves the (2,2) zero block of $\mathbf{H}$ in the computations.

1.4 Outline

This study is broken into three areas of focus.

The first area of focus is in chapters two and three. Chapter two begins with the linearization of QEP matrix into the companion matrix $\mathbf{A}$ and then applies a similarity transformation to form the skew-symmetric matrix $\mathbf{H}$. It first outlines the transformation of the QEP matrix into a standard eigenvalue problem of the previously defined companion matrix,

$$\mathbf{A} = \begin{pmatrix} -\mathbf{B} & -\mathbf{C} \\ \mathbf{I} & \mathbf{0} \end{pmatrix}.$$

Again, in this particular QEP, the constraints are that the submatrix **B** is skew-symmetric, the submatrix **C** is symmetric positive definite (SPD), and both of these matrices are n×n. Once the linearization of QEP into the companion matrix **A** is accomplished, the second step is to apply a similarity transformation of the companion matrix into the sparse skew symmetric structure denoted as matrix **H**. The physical and numerical motivation for constructing both matrix **A** and **H** are also discussed. Chapter three introduces some definitions and delineates some key properties of the Given's transformation matrices germane to this study.

In the fourth chapter, the focus is a numerical approach to diagonalizing the matrix **H** with specific constraints of leveraging the sparse structure of matrix **H**. Diagonalizing matrix **H** under these specific constraints is essentially the crux of the study. However, instead of fully diagonalizing matrix **H**, the numerical process is to conduct a real Schur factorization of matrix **H** which is essentially equivalent to diagonalizing the matrix **H** and thus matrix **A** as well. A real Schur factorization of a matrix provides the same information as diagonalizing the matrix which defines the normal modes. Chapter four's steps are as follows:

i) Theoretically show that matrix **H** can be transformed into a real Schur form.

ii) Determine a numerical algorithm which transforms **H** in to the Schur form while leveraging both the skew-symmetric and sub-matrix **0** sparse structures.

In our last area of focus, chapter five analyzes the algorithm (the algorithm is hence forth denoted as the "The Method") efficiency and accuracy. The analysis does an estimate of the flops required to execute The Method and compares that to a standard QR algorithm flop cost. Then, it explores the theoretical $\Delta\lambda$ error

bound relation with The Method's computed eigenvalues.

From this study on this specific QEP in the matrix form **H**, The Method's computational costs is $\frac{8n^3}{3}$ faster than the symmetric QR eigenvalue finding algorithm; and the eigenvalue errors appear to be generally bounded which implies accuracy in the algorithm. In addition, for future perturbation sensitivity analysis, The Method preserves the quadratic eigenvalue problem structure throughout most of the algorithms process. This preservation of QEP structure in the perturbed matrix **H** has a potential for determining a backward error bound on the calculated eigenvalue error.

Chapter 2

QEP's companion matrix & Its Skew-Symm. Variation

2.1 A companion matrix of the QEP's and its interpretation in ODE's

The quadratic eigenvalue problem (1.1) is a quadratic equation which requires linearization to a companion matrix in order to find its eigenvalues using the numerical eigenvalue finding techniques. However, "linearization is not unique and it is important to choose one that respects the symmetry and other structural properties..."[6]. In this study, we will choose first a classical linearization of

$$\left(\lambda^2\mathbf{I} + \lambda\mathbf{B} + \mathbf{C}\right)\mathbf{x} = 0. \tag{2.1}$$

The linearization of (2.1) constructs a companion matrix $\mathbf{A}$ and vector $\mathbf{y}$ such that

$$\left(\lambda\mathbf{I} - \mathbf{A}\right)\mathbf{y} = 0 \tag{2.2}$$

or equivalently $\mathbf{A}\mathbf{y} = \lambda\mathbf{y}$, where

$$\mathbf{A} = \begin{pmatrix} -\mathbf{B} & -\mathbf{C} \\ \mathbf{I} & \mathbf{O} \end{pmatrix} \quad \text{and} \quad \mathbf{y} = \begin{pmatrix} \lambda\mathbf{x} \\ \mathbf{x} \end{pmatrix}$$

Lemma 1: The linearization of QEP (2.2) is equivalent to (1.1).

Proof:

$$\begin{aligned}
\lambda^2\mathbf{x} + \lambda\mathbf{B}\mathbf{x} + \mathbf{C}\mathbf{x} = 0 \qquad & \Longleftrightarrow \\
\begin{pmatrix} -\lambda\mathbf{B}\mathbf{x} - \mathbf{C}\mathbf{x} \\ \lambda\mathbf{x} + 0 \end{pmatrix} = \begin{pmatrix} \lambda^2\mathbf{x} \\ \lambda\mathbf{x} \end{pmatrix} \qquad & \Longleftrightarrow \\
\begin{pmatrix} -\mathbf{B} & -\mathbf{C} \\ \mathbf{I} & 0 \end{pmatrix} \begin{pmatrix} \lambda\mathbf{x} \\ \mathbf{x} \end{pmatrix} = \begin{pmatrix} \lambda^2\mathbf{x} \\ \lambda\mathbf{x} \end{pmatrix} \qquad & \Longleftrightarrow \\
\begin{pmatrix} -\mathbf{B} & -\mathbf{C} \\ \mathbf{I} & 0 \end{pmatrix} \begin{pmatrix} \lambda\mathbf{x} \\ \mathbf{x} \end{pmatrix} = \lambda \begin{pmatrix} \lambda\mathbf{x} \\ \mathbf{x} \end{pmatrix} \qquad & QED
\end{aligned} \tag{2.3}$$

Clearly, (λ, $\mathbf{x}$) is an eigenpair of QEP if and only if (λ, $\mathbf{y}$) is an eigenpair of $\mathbf{A}$, the linearization of QEP. The length of the eigenvector $\mathbf{y}$ is doubled compared to $\mathbf{x}$, the QEP's eigenvector. As previously noted, this special linearization, $\mathbf{A}$, is conventionally known as a companion matrix for (2.1). Numerically seeking to transform the QEP into a linearized form is desirable. Since for a linearization like $\mathbf{A}$, there are already many well established eigenvalue finding algorithms available to apply.

Matrix $\mathbf{A}$ has a sparse structure with a zero submatrix in the (2,2) block and the identity submatrix in the (2,1) block. In addition, the constraints ensure that submatrix $\mathbf{B}$ in the (1,1) block is skew-symmetric and $\mathbf{C}$ in the (1,2) block is symmetric positive definite. However, in order to numerically exploit more structures, this study will transform matrix $\mathbf{A}$ further to $\mathbf{H}$, a skew-symmetric matrix. When

transforming matrix **A** to matrix **H**, the transformation seeks to preserve matrix **A**'s sparse structure while enhancing the skew symmetric structure throughout the entire matrix rather than just locally in sub-matrix **B**. This new matrix structure lends itself for a new numerical eigenvalue finding algorithm which capitalizes on both its sparsity and skew-symmetric structure.

2.2 Transformation of A into H

Matrix **A** is transformed into a skew-symmetric matrix designated as matrix **H**. The aim is to preserve matrix **A**'s eigenvalues structure but transform the matrix into a structure favorable to numerical methods for a Schur decomposition. Hence, all transformations of **A** to **H** must be similarity transformations. Matrix **H** also inherits the (2,2) block **0**.

The Cholesky-like factorization of matrix **C** is the key component in constructing a similarity transformation of matrix **A** to an entire skew-symmetric matrix. Since matrix **C** is a symmetric positive definite matrix, it has a Cholesky factorization $\mathbf{C}=\mathbf{L}\mathbf{L}^{\mathbf{T}}$, where **L** is a nonsingular lower triangular and $\mathbf{L}^{\mathbf{T}}$ is upper triangular [5]. But, there also exists a Cholesky-like factorization such that $\mathbf{C}=\mathbf{R}\mathbf{R}^{\mathbf{T}}$, where **R** is a nonsingular upper triangular and $\mathbf{R}^{\mathbf{T}}$ is lower triangular. See Appendix A for how this is developed. Utilizing this Cholesky-Like factorization property, a similarity transformation is constructed which can transform the eigenvalue problem $\mathbf{A}\mathbf{y} = \lambda\ \mathbf{y}$ into $\mathbf{H}\mathbf{u} = \lambda\mathbf{u}$.

Lemma 2: $\mathbf{H} = \mathbf{SAS^{-1}}$ where $\mathbf{H} = \begin{pmatrix} \mathbf{-B} & \mathbf{-R} \\ \mathbf{R^T} & \mathbf{0} \end{pmatrix}$ and $\mathbf{u} = \begin{pmatrix} \lambda\mathbf{x} \\ \mathbf{R^T\,x} \end{pmatrix}$.

Proof:

$$\mathbf{S}=\begin{pmatrix} \mathbf{I} & \mathbf{O} \\ \mathbf{O} & \mathbf{R^T} \end{pmatrix} \implies \mathbf{S^{-1}} = \begin{pmatrix} \mathbf{I} & \mathbf{O} \\ \mathbf{O} & \mathbf{R^{-T}} \end{pmatrix}.$$

Then

$$\begin{aligned} & (\lambda\mathbf{I} - \mathbf{A})\,\mathbf{y} = 0 \\ \iff\ & \mathbf{Ay} = \lambda\mathbf{y} \\ \iff\ & \mathbf{SAy} = \lambda\mathbf{S\ y} \\ \iff\ & \mathbf{S\ A\ S^{-1}\ S\ y} = \lambda\mathbf{S\ y} \\ \iff\ & \mathbf{Hu} = \lambda\mathbf{u}. \end{aligned}$$

$$\text{So } \mathbf{H} = \mathbf{SAS^{-1}} = \begin{pmatrix} \mathbf{I} & \mathbf{O} \\ \mathbf{O} & \mathbf{R^T} \end{pmatrix} \begin{pmatrix} \mathbf{-B} & \mathbf{-RR^T} \\ \mathbf{I} & \mathbf{O} \end{pmatrix} \begin{pmatrix} \mathbf{I} & \mathbf{O} \\ \mathbf{O} & \mathbf{R^{-T}} \end{pmatrix}$$

$$= \begin{pmatrix} \mathbf{-B} & \mathbf{-R} \\ \mathbf{R^T} & \mathbf{O} \end{pmatrix}.$$

Therefore, $\mathbf{u}=\mathbf{S\ y}=\begin{pmatrix} \mathbf{I} & \mathbf{O} \\ \mathbf{O} & \mathbf{R^T} \end{pmatrix} \begin{pmatrix} \lambda\mathbf{x} \\ \mathbf{x} \end{pmatrix} = \begin{pmatrix} \lambda\mathbf{x} \\ \mathbf{R^T\,x} \end{pmatrix}$ and $\mathbf{H} = \begin{pmatrix} \mathbf{-B} & \mathbf{-R} \\ \mathbf{R^T} & \mathbf{O} \end{pmatrix}$

is a skew-symmetric matrix which is similar to the original companion matrix $\mathbf{A}$. QED.

Note that the (2,2) block of **H** remains zero. Moreover, **H** is characterized with the skew-symmetric matrix **B** and the upper triangular matrix **R**.

Chapter 3

Definitions and the Givens Matrices

3.1 Proposition and Definitions

Definition 1 Skew-Symmetric Matrices*: $\mathbf{S} \in \mathbb{R}^{n\times n}$ is skew-symmetric if $\mathbf{S}^T = -\mathbf{S}$.*

Definition 2 Similarity transformations*: Suppose $\mathbf{F}$ is nonsingular, then the transformation $\mathbf{D} = \mathbf{FMF}^{-1}$ from $\mathbf{M}$ to $\mathbf{D}$ is a similarity transformation. $\mathbf{F}$ is called a similarity transformation matrix [5].*

Definition 3 Congruence transformations*: Suppose $\mathbf{F}$ is nonsingular, then the transformation $\tilde{\mathbf{H}} = \mathbf{F^T HF}$ from $\mathbf{H}$ to $\tilde{\mathbf{H}}$ is a congruence transformation. $\mathbf{F}$ is called a congruence transformation matrix.*

Definition 4 Orthogonal Matrices*: A real square matrix $\mathbf{Q}$ is orthogonal if $\mathbf{QQ^T} = \mathbf{I}$.*

Proposition 3.1.1 Similarity Congruence transformations*: If* $\mathbf{Q}$ *is real orthogonal matrix, then the similarity transformation* $\mathbf{H} \rightarrow \mathbf{Q}^{-1}\mathbf{H}\mathbf{Q}$ *is also a congruence transformation.*

From definition 1, for a skew-symmetric matrix $\mathbf{S}$, all its eigenvalues are imaginary and all its diagonal elements are zero. From definition 2, if (λ, $\mathbf{z}$) is an eigenpair of matrix $\mathbf{D}$, where λ is the eigenvalue and $\mathbf{z}$ is its right eigenvector, then (λ, $\mathbf{F}^{-1}\mathbf{z}$) is an eigenpair of matrix $\mathbf{M}$. From definition 3, congruence transformations preserve skew-symmetric and symmetric structures of matrices. For instance, if $\mathbf{H}$ is a skew-symmetric matrix, then so is $\tilde{\mathbf{H}}$. Finally, from definition 4, for a real orthogonal matrix $\mathbf{Q}$, the inner-product is preserved $\langle \mathbf{Q}\mathbf{y}, \mathbf{Q}\mathbf{x} \rangle = \mathbf{y}^{\mathbf{T}}\mathbf{Q}^{\mathbf{T}}\mathbf{Q}\mathbf{x} = \mathbf{y}^{\mathbf{T}}\mathbf{x} = \langle \mathbf{y}, \mathbf{x} \rangle$, i.e. length is preserved under orthogonal transformations.

3.2 Givens matrices

Givens matrices, also known as Givens rotations, are orthogonal matrices. Hence, similarity transformations with Givens matrices preserve both the eigenvalues and the structure of a skew-symmetric matrix. Givens matrices in their smallest form are 2x2 orthogonal matrices which can rotate a vector in a plane. Any 2×2 Givens matrix can be denoted as

$$\mathbf{G} = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} = \begin{pmatrix} c & -s \\ s & c \end{pmatrix} \tag{3.1}$$

where $c = \cos\theta$ and $s = \sin\theta$. Note that $\mathbf{G}^{\mathbf{T}}$ is also a Givens matrix.

For any $\mathbf{x} \in \mathbb{R}^2$,

$$\tilde{\mathbf{x}} = \mathbf{G^T x} = \mathbf{G^T} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} cx_1 + sx_2 \\ -sx_1 + cx_2 \end{pmatrix}$$

is the vector generated by rotating the $\mathbf{x}$ counter-clockwise with the angle θ in the $\mathbb{R}^2$ plane.

Certainly, one can always determine a Givens matrix which can rotate a given vector to a vector that is parallel to one of the coordinate base vectors. Based on the vector's configuration and type of rotations, there are four basic Givens rotations.

- To rotate the column vector $\mathbf{x}=\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ parallel to $\mathbf{e_2} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, i.e.

$$\mathbf{G^T} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} cx_1 + sx_2 \\ -sx_1 + cx_2 \end{pmatrix} = \begin{pmatrix} 0 \\ \tilde{x_2} \end{pmatrix}.$$

  Then c and s should satisfy $cx_1 + sx_2 = 0$. With c = $\cos\theta$ and s = $\sin\theta$ or the constraints: $c^2 + s^2 = 1$, one chooses

$$c = \frac{x_2}{\sqrt{x_1^2+x_2^2}},\ s = \frac{-x_1}{\sqrt{x_1^2+x_2^2}} \quad \text{or simply} \quad \theta = \arctan(-x_1/x_2).$$

- To rotate the column vector $\mathbf{x}$ parallel to the $\mathbf{e_1} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, i.e.,

$$\mathbf{G^T} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} cx_1 + sx_2 \\ -sx_1 + cx_2 \end{pmatrix} = \begin{pmatrix} \tilde{x}_1 \\ 0 \end{pmatrix}.$$

  Then c and s should satisfy $-sx_1 + cx_2 = 0$. With c = $\cos\theta$ and s = $\sin\theta$ or the constraints: $c^2 + s^2 = 1$, one chooses

$$c = \frac{x_1}{\sqrt{x_1^2+x_2^2}},\ s = \frac{x_2}{\sqrt{x_1^2+x_2^2}} \text{ or simply } \theta = \arctan(x_2/x_1).$$

,

- To rotate the row vector $\mathbf{x} = (x_1, x_2)$ parallel to the $\mathbf{e_1^T} = \begin{pmatrix} 1 & 0 \end{pmatrix}$, i.e.,

$$\begin{pmatrix} x_1 & x_2 \end{pmatrix} \mathbf{G} = \begin{pmatrix} cx_1 + sx_2 & -sx_1 + cx_2 \end{pmatrix} = \begin{pmatrix} \tilde{x}_1 & 0 \end{pmatrix}.$$

  Then c and s should satisfy $-sx_1 + cx_2 = 0$. With c = $\cos\theta$ and s = $\sin\theta$ or the constraints: $c^2 + s^2 = 1$, one chooses

$$c = \frac{x_1}{\sqrt{x_1^2+x_2^2}},\ s = \frac{x_2}{\sqrt{x_1^2+x_2^2}}, \text{ or simply } \theta = \arctan(x_2/x_1).$$

- To rotate the row vector $\mathbf{x}$ parallel to the $\mathbf{e_2^T} = \begin{pmatrix} 0 & 1 \end{pmatrix}$, i.e.

$$\begin{pmatrix} x_1 & x_2 \end{pmatrix} \mathbf{G} = \begin{pmatrix} cx_1 + sx_2 & -sx_1 + cx_2 \end{pmatrix} = \begin{pmatrix} 0 & \tilde{x_2} \end{pmatrix}.$$

  Then c and s should satisfy $cx_1 + sx_2 = 0$. With c = $\cos\theta$ and s = $\sin\theta$ or the constraints: $c^2 + s^2 = 1$, one chooses

$$c = \frac{x_2}{\sqrt{x_1^2+x_2^2}},\ s = \frac{-x_1}{\sqrt{x_1^2+x_2^2}}, \text{ or simply } \theta = \arctan(-x_1/x_2).$$

Therefore, with any 2 x 2 matrix, a Givens matrix can zero one of the matrices' entry (i.e. shift the mass from a specific element to an adjacent one on the same row or column, depending on whether the Given's is multiplied from the left or right. Note, for the purpose of clarity, this study may simply state that a Givens matrix $\mathbf{G}$ is used to rotate some row or column vector. If the vector is a column vector, the reader can assume that $\mathbf{G^T}$ is applied from the left; and if it's a row vector then $\mathbf{G}$

is applied from the right. Below, are some typical examples in applying the Givens matrices:

$$\begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} \mathbf{G} = \overset{\rightarrow}{\begin{pmatrix} 0 & \tilde{\mathbf{x}}_{\mathbf{12}} \\ \tilde{x}_{21} & \tilde{x}_{22} \end{pmatrix}}$$

$$\mathbf{G^T} \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} = \overset{\downarrow}{\begin{pmatrix} 0 & \tilde{x}_{12} \\ \tilde{\mathbf{x}}_{\mathbf{21}} & \tilde{x}_{22} \end{pmatrix}}$$

$$\begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} \mathbf{G} = \overset{\leftarrow}{\begin{pmatrix} \tilde{x}_{11} & \tilde{x}_{12} \\ \tilde{\mathbf{x}}_{\mathbf{21}} & 0 \end{pmatrix}}$$

$$\mathbf{G^T} \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} = \overset{\uparrow}{\begin{pmatrix} \tilde{\mathbf{x}}_{\mathbf{11}} & \tilde{x}_{12} \\ 0 & \tilde{x}_{21} \end{pmatrix}}$$

When applying a Givens matrix to matrix $\mathbf{A} \in \mathbf{R}^{n \times n}$, the Givens matrix takes

the form

$$\mathbf{G}_{ij}(c,s) = \begin{array}{c} \\ 1 \\ \\ i \\ \\ \\ \\ j \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccc} & & i & & & j & \end{array} \\ \begin{pmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & c & & & -s & \\ & & & \ddots & & & \\ & & & & 1 & & \\ & & & & & \ddots & \\ & & s & & & c & \\ & & & & & & \ddots \\ & & & & & & & 1 \end{pmatrix}_{n\mathbf{x}n} \end{array} \tag{3.2}$$

The matrix product $\mathbf{AG_{ij}}(c,s)$ means that the Givens matrix $\begin{pmatrix} c & -s \\ s & c \end{pmatrix}$ acts on columns i and j of $\mathbf{A}$. Similarly, $\mathbf{G_{ij}^T}(c,s)\mathbf{A}$ means that the Givens matrix $\begin{pmatrix} c & -s \\ s & c \end{pmatrix}$ acts on rows i and j of $\mathbf{A}$. This means one can design a Givens matrix, $\mathbf{G_{ij}}(c,s)$ to zero any entry in an effected column(row) and shift its mass to the corresponding entry in another effected column(row). For simplicity, this study will use $\mathbf{G_{ij}}$ in lieu of $\mathbf{G}_{ij}(c,s)$ unless c,s needs to be emphasized. The applications of Givens matrices are illustrated on a 5×5 matrices with $i = 2$ and $j = 4$

$$\mathbf{G_{24}^T} \begin{pmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{pmatrix} \quad \mapsto \quad \begin{array}{c} \\ 2 \\ \\ 4 \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccc} & & & & \end{array} \\ \begin{pmatrix} x & x & x & x & x \\ \otimes & \otimes & \otimes & \otimes & \otimes \\ x & x & x & x & x \\ \otimes & \otimes & \otimes & \otimes & \otimes \\ x & x & x & x & x \end{pmatrix} \end{array}$$

$$\begin{pmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{pmatrix} \mathbf{G}_{24} \quad \mapsto \quad \begin{matrix} & & 2 & & 4 \\ \end{matrix} \begin{pmatrix} x & \otimes & x & \otimes & x \\ x & \otimes & x & \otimes & x \\ x & \otimes & x & \otimes & x \\ x & \otimes & x & \otimes & x \\ x & \otimes & x & \otimes & x \end{pmatrix}$$

where the symbol $\otimes$ represents an entry affected by the Given's matrix $\mathbf{G}_{24}$.

Since Givens transformations are used extensively in this study, some notations are required to clarify the transformation effects. A Given's matrix operates on two vectors. When applying Givens matrices, its purpose is to zero out one of the components of a vector and transfer its mass to the other vector's corresponding component. To illustrate this, $\mathbf{G_{24}}$ is applied to a 5×5 matrix to zero out the x_{21} and shift that entry's mass to x_{41} entry.

$$\mathbf{G_{24}^T} \begin{pmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{pmatrix} \quad \mapsto \quad \begin{matrix} \\ 2 \\ \\ 4 \\ \\ \end{matrix} \begin{pmatrix} x & x & x & x & x \\ 0_{\downarrow} & \otimes & \otimes & \otimes & \otimes \\ x & x & x & x & x \\ \triangle & \otimes & \otimes & \otimes & \otimes \\ x & x & x & x & x \end{pmatrix}$$

If we apply $\mathbf{G_{24}}$ to a 5×5 matrix to zero out the x_{12} and shift that entry's mass to x_{14} entry.

$$\begin{pmatrix} x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \\ x & x & x & x & x \end{pmatrix} \mathbf{G_{24}} \quad \mapsto \quad \begin{matrix} & & 2 & & 4 & \\ & \begin{pmatrix} x & \overrightarrow{0} & x & \triangle & x \\ x & \otimes & x & \otimes & x \\ x & \otimes & x & \otimes & x \\ x & \otimes & x & \otimes & x \\ x & \otimes & x & \otimes & x \end{pmatrix} \end{matrix}$$

In the first example, the entry which was zeroed is denoted with the symbol, $0_{\downarrow}$. In the second example, the entry which was zeroed was denoted with the symbol, $\overrightarrow{0}$. The arrow in the zeroed symbol denotes in which direction the mass from the zeroed entry was transferred to. There are four possible symbols which denote an entry being zeroed: $0_{\downarrow}$, $0^{\uparrow}$, $\overrightarrow{0}$, $\overleftarrow{0}$. The mass transfered from the zeroed entry is deposited into the entry denoted as $\triangle$. All other entries not acted on by the Givens' transform remain as x entries. Except for the zeroed entry, 0, and the entry receiving the mass transfer, $\triangle$, all other entries affected by the Givens transformation are denoted as $\otimes$.

Symmetry effects occur when a Given's transformation is applied to a skew-symmetric matrix. By proposition 3.1.1, def. 2, and def. 3, the congruence transformation $\mathbf{G_{ij}^T\ H\ G_{ij}}$ with the Givens matrix $\mathbf{G_{ij}}$, preserves matrix $\mathbf{H}$'s skew-symmetric structure. Below, this study will discuss how the entries, h_{ij}, change under the transform $\mathbf{G_{ij}^T\ H\ G_{ij}}$. Suppose $\mathbf{H}$ is a 2×2 matrix such that

$$\mathbf{H} = \begin{pmatrix} 0 & x_{12} \\ x_{21} & 0 \end{pmatrix} = \begin{pmatrix} 0 & x_b \\ -x_b & 0 \end{pmatrix} \tag{3.3}$$

where $x_{12} = -x_{21} = x_b$. Then a congruence transformation of a 2×2 skew-symmetric matrix using a Given's transformation will preserve the skew-symmetric structure by not changing the matrix. To illustrate this, a Givens matrix is applied on the left side of matrix $\mathbf{H}$.

$$\mathbf{G^T}\begin{pmatrix} 0 & x_b \\ -x_b & 0 \end{pmatrix} = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}\begin{pmatrix} 0 & x_b \\ -x_b & 0 \end{pmatrix} = \begin{pmatrix} -x_b s & x_b c \\ -x_b c & -x_b s \end{pmatrix}$$

Now, applying the Givens from the right, the constraint $\cos^2\theta + \sin^2\theta = 1$, $(c^2 + s^2 = 1)$, returns the final transformed matrix back to its original skew-symmetric structure.

$$\mathbf{G^T}\begin{pmatrix} 0 & x_b \\ -x_b & 0 \end{pmatrix}\mathbf{G} = \begin{pmatrix} -x_b s & x_b c \\ -x_b c & -x_b s \end{pmatrix}\begin{pmatrix} c & -s \\ s & c \end{pmatrix} =$$

$$\begin{pmatrix} -x_b cs + x_b cs & x_b s^2 + x_b c^2 \\ -x_b c^2 - x_b s^2 & x_b cs - x_b cs \end{pmatrix} = \begin{pmatrix} 0 & x_b \\ -x_b & 0 \end{pmatrix}. \tag{3.4}$$

When applied to larger skew-symmetric matrices, this transformation not only does not change the 2×2 entries but it also preserves the skew-symmetric matrices as a whole. Let us illustrate this with a 4×4 matrix.

$$\mathbf{H} = \begin{pmatrix} 0 & h_{12} & h_{13} & h_{14} \\ h_{21} & 0 & h_{23} & h_{24} \\ h_{31} & h_{32} & 0 & h_{34} \\ h_{41} & h_{42} & h_{43} & 0 \end{pmatrix}.$$

Since $\mathbf{H}$ is skew-symmetric, then $h_{ij} = -h_{ji}$. If we apply a Givens transformation on this $\mathbf{H}$ matrix's 2 and 4 (column and row) entries, the Givens matrix would then have the following structure

$$\mathbf{G_{24}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & c & 0 & -s \\ 0 & 0 & 1 & 0 \\ 0 & s & 0 & c \end{pmatrix}.$$

Then, when applying the congruence transformation, we have

$$\widehat{\mathbf{H}} \equiv \mathbf{G_{24}^T}\ \mathbf{H}\ \mathbf{G_{24}}, \text{ then}$$

$$\widehat{\mathbf{H}} = \begin{array}{c} \\ 2 \\ \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} \quad & 2 & \quad & 4 \end{array} \\ \begin{pmatrix} 0 & ch_{12} + sh_{14} & h_{13} & ch_{14} - sh_{12} \\ ch_{21} + sh_{41} & csh_{24} + csh_{42} & ch_{23} + sh_{43} & c^2h_{24} - s^2h_{42} \\ h_{31} & ch_{32} + sh_{34} & 0 & ch_{34} - sh_{32} \\ ch_{41} - sh_{21} & c^2h_{42} - s^2h_{24} & ch_{43} - sh_{23} & -csh_{24} - csh_{42} \end{pmatrix} \end{array} = \hat{h}_{ij}$$

Since $h_{24} = -h_{42}$ then $\hat{h}_{22} = \hat{h}_{44} = 0$, $\hat{h}_{24} = h_{24}$ and $\hat{h}_{42} = h_{42}$, we can see immediately that the entries of (2,2), (2,4), (4,2), and (4,4) do not change in the transformation $\mathbf{G_{24}^T}\ \mathbf{H}\ \mathbf{G_{24}}$. These entries have the same structure as the 2×2 matrix depicted in (3.3) and (3.4). This submatrix $\begin{pmatrix} 0 & h_{24} \\ h_{42} & 0 \end{pmatrix}$ is therefore invariant. Likewise, we can see that all entries of matrix $\widehat{\mathbf{H}}$ satisfy $h_{ij} = -h_{ji}$. Therefore, the Givens transformations preserves any n×n skew-symmetric matrix and leaves the 4 entries corresponding to the c and s entries of the Givens matrix unchanged. In this study, these four entries are called the invariant entries of the Givens congruence transformations.

These invariant entries correspond to (3.3) which form a 2×2 submatrix within matrix $\mathbf{H}$ and is depicted as

$$\begin{pmatrix} \odot & \odot \\ \odot & \odot \end{pmatrix}. \tag{3.5}$$

In larger n×n, $\mathbf{H}$ matrix, the invariant entries h_{ii}, h_{ij}, h_{ji}, and h_{jj} corresponding to a Givens congruence transformation $\mathbf{G_{ij}}$ using $\mathbf{i}, \mathbf{j}$ are graphically depicted as follows.

$$
\begin{array}{c}
 \\ i \\ \\ \\ \\ \\ \\ \\ j \\ \\ \\ \\
\end{array}
\begin{array}{c}
\begin{array}{cccccccccc} & & & i & & & & j & & & \end{array} \\
\left(\begin{array}{cccccccccc}
0 & \vdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \vdots \\
\vdots & \ddots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \vdots \\
\vdots & \vdots & 0 & \vdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \vdots \\
\vdots & \cdots & \vdots & \odot & \vdots & \cdots & \vdots & \odot & \vdots & \cdots & \vdots \\
\vdots & \cdots & \cdots & \vdots & 0 & \vdots & \cdots & \cdots & \cdots & \cdots & \vdots \\
\vdots & \vdots & \vdots & \cdots & \cdots & \ddots & \cdots & \cdots & \cdots & \cdots & \vdots \\
\vdots & \cdots & \vdots & \vdots & \cdots & \vdots & 0 & \vdots & \cdots & \cdots & \vdots \\
\vdots & \cdots & \cdots & \odot & \cdots & \cdots & \cdots & \odot & \vdots & \cdots & \cdots \\
\vdots & \cdots & \vdots & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & \vdots & \vdots \\
\vdots & \cdots & \cdots & \cdots & \cdots & \cdots & \vdots & \vdots & \cdots & \ddots & \vdots \\
\vdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \vdots & 0
\end{array}\right)
\end{array}
$$

The four invariant entries, $\odot$, also form an invariant submatrix. This property is essential later when numerically transforming $\mathbf{H}$ into the real Schur form while simultaneously preserving (not contaminating) the zero submatrix.

Chapter 4

Diagonalization of matrix H

This chapter analyzes the numerical processes in transforming matrix $\mathbf{H}$ into the Schur form which is equivalent to diagonalizing matrix $\mathbf{H}$. Transforming matrix $\mathbf{A}$ into a skew-symmetric $\mathbf{H}$ allows an exploitation of anti-symmetry and numerical stability along with preserving the original zero sub-matrix. So, it enhances both numerical efficiency and physical relevance in determining the eigenvalues. Since orthogonal similarity transformations preserves both eigenvalue and matrix structures, it is now possible, due to the skew-symmetric properties and the $\mathbf{0}$ sub matrix structure, to just concentrate numerically on either the upper half of the $\mathbf{H}$ matrix $\begin{pmatrix} -\mathbf{B} & -\mathbf{R} \end{pmatrix}$ or the left half $\begin{pmatrix} -\mathbf{B} \\ \mathbf{R^T} \end{pmatrix}$. By convention, this study will focus on the upper half of matrix $\mathbf{H}$. Likewise, if an algorithm can leveraging the zero-submatrix, then unnecessary computational costs can potentially be eliminated or reduced. So, the skew-symmetric structure and the sub-matrix zero structures are critical in saving operational cost in converting $\mathbf{H}$ to a real Schur form.

Secondly, the form of both matrices $\mathbf{A}$ and $\mathbf{H}$ preserves the basic structure of the original QEP equation (2.1) $\left(\lambda^2\mathbf{I} + \lambda\mathbf{B} + \mathbf{C}\right)\mathbf{X} = 0$. If all the subsequent transfor-

mations preserve the (2,2) zero block in **H**, then every intermediate matrix during the transformations has an equivalent QEP. This shows the possibility that the computed eigenvalues can be formulated as the exact ones of a slightly perturbed QEP from the original QEP. Note that currently no method can provide such eigenvalues from the perturbed QEP.

4.1 Schur decomposition of H

In this study, Schur decomposition of matrix **H** is equivalent to diagonalizing matrix **H**.

$$\mathbf{H} \;\rightarrow\; \mathbf{T} \;\rightarrow\; \left\{ \begin{array}{l} \rightarrow \;\; \mathbf{D} \\ \\ \rightarrow \;\; \widehat{\mathbf{T}} \end{array} \right. \tag{4.1}$$

where **T** is the Schur form of **H** which is essentially equivalent to **D** the diagonalization of **H** or $\widehat{\mathbf{T}}$ a special similarity form of **H**.

Every real square matrix has a Schur decomposition [5, 7]. A Schur decomposition of a square matrix **H** is defined as

$$\mathbf{H} = \mathbf{Q}\ \mathbf{T}\ \mathbf{Q}^{\mathbf{T}} \;\;\Longrightarrow\;\; \mathbf{T} = \mathbf{Q}^{\mathbf{T}}\ \mathbf{H}\ \mathbf{Q} \tag{4.2}$$

where **Q** is a real orthogonal and **T** is quasi-upper-triangular matrix called the Schur form. With the transformation **Q** being an orthogonal matrix, the decomposition is a congruence similarity transformation. Therefore, when **H** is real skew-symmetric, **T** has to be real skew-symmetric. With a real symmetric structure, **T** is then a block diagonal with block sizes of 1×1 or 2×2. For a nonsingular $m \times m$ skew-symmetric

matrix, m is necessarily even and the real Schur decomposition takes a 2×2 block diagonal form

$$\mathbf{T} = \begin{pmatrix} 0 & t_1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ -t_1 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & t_2 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & -t_2 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & t_3 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & -t_3 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ddots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & t_n \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & -t_n & 0 \end{pmatrix} \tag{4.3}$$

where we assume here that $m = 2n$.

For a 2×2 nonsingular skew-symmetric matrix, the $\mathbf{T}$ real Schur decomposition is just one 2×2 block diagonal

$$\mathbf{T} = \begin{pmatrix} 0 & t_1 \\ -t_1 & 0 \end{pmatrix}, \qquad t_1 \neq 0 \tag{4.4}$$

In determining the eigenvalues of this $\mathbf{T}$, one uses the similarity transformation matrix $\mathbf{Z} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ i & -i \end{pmatrix}$. Then

$$\mathbf{D} = \mathbf{Z}^* \begin{pmatrix} 0 & t_1 \\ -t_1 & 0 \end{pmatrix} \mathbf{Z} = \begin{pmatrix} \imath\, t_1 & 0 \\ 0 & -\imath\, t_1 \end{pmatrix} \tag{4.5}$$

Applying the transformation (4.5) to every 2×2 diagonal block in (4.3) diagonalizes the quasi-block real Schur matrix $\mathbf{T}$ into the form

$$\mathbf{D} = \begin{pmatrix} \imath\, t_1 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & -\imath\, t_1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \imath\, t_2 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & -\imath\, t_2 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & \imath\, t_3 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -\imath\, t_3 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \ddots & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & \imath\, t_n & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 & -\imath\, t_n \end{pmatrix} \tag{4.6}$$

It is then clear, that the real Schur form in the form of (4.3) is equivalent to the diagonal matrix **D** in (4.6). The eigenvalues of **T** are pure imaginary as expected for a skew-symmetric matrix. Then, for the diagonalization of (4.3), its diagonal entries (eigenvalues) are also pure imaginary. In this study, the Schur decomposition of matrix **H** into the real Schur form will not be in the quasi-diagonal form of (4.3), but by convention, in a real form as depicted:

$$\hat{\mathbf{T}} = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 & \hat{t}_1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & \hat{t}_2 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 0 & 0 & 0 & \hat{t}_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & \hat{t}_n \\ -\hat{t}_1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & -\hat{t}_2 & 0 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & -\hat{t}_3 & \cdots & 0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & \cdots & -\hat{t}_n & 0 & 0 & 0 & \cdots & 0 \end{pmatrix} \tag{4.7}$$

$\hat{\mathbf{T}}$ is similar to $\mathbf{T}$. The permutation orthogonal transformation matrix, $\mathbf{P} = [e_1, e_{n+1}, e_2, e_{n+2}, \cdots, e_n, e_{2n}]$, which is orthogonal, is a similarity transformation between (4.3) and (4.7).

$$\mathbf{T} = \mathbf{P^T}\ \hat{\mathbf{T}}\ \mathbf{P} \tag{4.8}$$

So, we may also consider (4.7) as a real Schur form of $\mathbf{H}$. In this study, the aim is to diagonalize matrix $\mathbf{H}$ into the real Schur form of matrix $\hat{\mathbf{T}}$, (4.7).

4.2 Numerical approach for a Real Schur factorization of matrix H

The focus is on a numerical approach which conducts a real Schur decomposition (4.7) of matrix $\mathbf{H}$ without contaminating its (2,2) zero block nor destroying its skew-symmetric symmetry. Although there are many ways to diagonalize or equivalently transform a matrix into a real Schur form, this study is analyzing a particular numerical approach designated as The Method.

Description of the Algorithm: The Method's algorithm leverages both the (2,2) zero block and the skew-symmetry to reduce computational decomposition costs of **H**. Using the Given's similarity-congruence transformations, The Method's iterative transformation zeros out the **B** submatrix and reduces **R** (and $\mathbf{R^T}$) to an upper and lower bidiagonal respectively row by row and column by column. When zeroing out **B**, it ultimately shifts **B**'s mass to both **R** and $\mathbf{R^T}$ submatrices. Simultaneously, both **R** and $\mathbf{R^T}$ are reduced to an upper and lower bidiagonal matrix, respectively. This transformation algorithm is done without contaminating the **0** submatrix nor destroying the skew-symmetry of **H**. For notational convenience, we will consider the reduction on

$$-\mathbf{H} = \begin{pmatrix} \mathbf{B} & \mathbf{R} \\ -\mathbf{R^T} & 0 \end{pmatrix}$$

Essentially, The Method's algorithm iterations does the following;

$$\mathbf{B} \longrightarrow \mathbf{0} \quad \text{and} \quad \mathbf{R} \longrightarrow \mathbf{U} \quad = \begin{pmatrix} u_{11} & u_{12} & & \\ & u_{22} & \ddots & \\ & & \ddots & u_{n-1,n} \\ & & & u_{nn} \end{pmatrix}$$

$$\mathbf{R^T} \longrightarrow \mathbf{U^T} \quad = \begin{pmatrix} u_{11} & & & \\ u_{12} & u_{22} & & \\ & \ddots & \ddots & \\ & & u_{n,n-1} & u_{nn} \end{pmatrix}$$

Once the bidiagonals are achieved, a SVD transformation is conducted for each submatrix upper bidiagonal to reduce matrix **H** into the real Schur form $\widehat{\mathbf{T}}$. Due to matrix **H**'s skew-symmetry, The Method's algorithm only has to be applied to the upper two blocks (1,1) and (1,2) of **H**. Operating only on the upper two blocks

prevents unnecessary flop operations due to skew-symmetry properties which then ultimately saves overall flop costs.

This study will now illustrate the process of reducing $\mathbf{B}$ to $\mathbf{0}$, $\mathbf{R}$ to an upper bidiagonal, and $\mathbf{R^T}$ to a lower bidiagonal submatrices through consecutive Givens transformations. Stage one begins with shifting the mass of the first row entries of $\mathbf{B}$ toward b_{1n} which is the last or nth entry of the first row. Due to the skew-symmetry of $\mathbf{H}$, this Givens congruence transformation also means that the mass located in the first column in submatrix $\mathbf{B}$ is also shifted toward b_{n1} which is the last or nth entry in the first column. In addition, these transformations have a mirror effect on both submatrices $\mathbf{R}$ and $\mathbf{R^T}$. The following illustration of the first stage shows how b_{12} is eliminated and its mass is shifted to b_{13} on an 8×8 matrix.

$$-\mathbf{H} = \begin{pmatrix} \mathbf{B} & \mathbf{R} \\ -\mathbf{R^T} & \mathbf{0} \end{pmatrix}$$

$$-\mathbf{H} = \begin{pmatrix} 0 & b_{12} & b_{13} & b_{14} & r_{11} & r_{12} & r_{13} & r_{14} \\ b_{21} & 0 & b_{23} & b_{24} & 0 & r_{22} & r_{23} & r_{24} \\ b_{31} & b_{32} & 0 & b_{34} & 0 & 0 & r_{33} & r_{34} \\ b_{41} & b_{42} & b_{43} & 0 & 0 & 0 & 0 & r_{44} \\ & & & & & & & \\ -r_{11} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -r_{12} & -r_{22} & 0 & 0 & 0 & 0 & 0 & 0 \\ -r_{13} & -r_{23} & -r_{33} & 0 & 0 & 0 & 0 & 0 \\ -r_{14} & -r_{24} & -r_{34} & -r_{44} & 0 & 0 & 0 & 0 \end{pmatrix} \hookrightarrow$$

$$\mathbf{G_{23}^T} \begin{pmatrix} \mathbf{B} & \mathbf{R} \\ -\mathbf{R^T} & \mathbf{0} \end{pmatrix} \mathbf{G_{23}} =$$

$$\begin{pmatrix} 0 & \overrightarrow{0} & \triangle & b_{14} & r_{11} & r_{12} & r_{13} & r_{14} \\ 0_{\downarrow} & 0 & b_{23} & \otimes & 0 & \otimes & \otimes & \otimes \\ \triangle & b_{32} & 0 & \otimes & 0 & \mathbf{x} & \otimes & \otimes \\ b_{41} & \otimes & \otimes & 0 & 0 & 0 & 0 & -r_{44} \\ \\ -r_{11} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -r_{12} & \otimes & \mathbf{x} & 0 & 0 & 0 & 0 & 0 \\ -r_{13} & \otimes & \otimes & 0 & 0 & 0 & 0 & 0 \\ -r_{14} & \otimes & \otimes & -r_{44} & 0 & 0 & 0 & 0 \end{pmatrix}$$

Since **H**'s skew-symmetry is preserved in these Givens transformations, any Givens' similarity transformation which collectively affects submatrix **R** and matrix **B** also affects collectively -$\mathbf{R^T}$ and matrix **B** symmetrically in the same manner. Rather than applying the algorithm to the whole matrix **H**, the algorithm only needs to be applied on the submatrices **R** and **B**. Any effects on matrix **R** from the Givens transformation are posted into the submatrix -$\mathbf{R^T}$. Therefore, it suffices to only calculate the computations for the upper two blocks of a $2n \times 2n$ size -**H** matrix. For $1 \leq$ i, j $\leq\ n$,

$$\mathbf{\tilde{G}_{ij}^T}\,[\mathbf{B},\mathbf{R}]\,\mathbf{G_{ij}} \Longleftrightarrow\ \mathbf{G_{ij}^T}\ \mathbf{H}\ \mathbf{G_{ij}} \tag{4.9}$$

$$[\mathbf{B},\mathbf{R}]\,\mathbf{G}_{n+i,n+j} \Longleftrightarrow\ \ \mathbf{G^T}_{n+i,n+j}\mathbf{H}\ \mathbf{G}_{n+i,n+j}$$

where $\mathbf{G_{ij}}$ is the principal $n \times n$ submatrix of the $2n \times 2n$ Givens matrix $\mathbf{G_{ij}}$ which both have the Givens matrix form of (3.2). All subsequent discussions of The Method, except when specified otherwise, will focus collectively on a 4×8, submatrix $[\mathbf{B},\mathbf{R}]$ the upper two block submatrices of -**H** with n =4.

For this study, the upper two submatrix blocks of matrix -**H** are notationally abbreviated as follows:

$$\mathbf{K} = [\mathbf{B}, \mathbf{R}] = \begin{pmatrix} 0 & b_{12} & b_{13} & b_{14} & r_{11} & r_{12} & r_{13} & r_{14} \\ b_{21} & 0 & b_{23} & b_{24} & 0 & r_{22} & r_{23} & r_{24} \\ b_{31} & b_{32} & 0 & b_{34} & 0 & 0 & r_{33} & r_{34} \\ b_{41} & b_{42} & b_{43} & 0 & 0 & 0 & 0 & r_{44} \end{pmatrix} \Longrightarrow$$

$$\mathbf{K} = \begin{pmatrix} 0 & x & x & x & x & x & x & x \\ x & 0 & x & x & 0 & x & x & x \\ x & x & 0 & x & 0 & 0 & x & x \\ x & x & x & 0 & 0 & 0 & 0 & x \end{pmatrix}$$

Continuing with stage one, The Method iteratively shifts the first row mass entries of submatrix **B** toward the last row entry, b_{14}, in submatrix **B**. Beginning with the first row entry b_{12}, its mass is shifted to the immediate entry to the right, b_{13}. Essentially, for each transformation iteration, the mass in $b_{1,j}$ is shifted to $b_{1,j+1}$ entry. However, for each iteration, contamination occurs in the zero entries of the **R** and must be cleaned up without contaminating the sub-matrix **0** or reintroducing the mass back into submatrix **B**.

$$\tilde{\mathbf{G}}_{\mathbf{23}}^{\mathbf{T}}\ \mathbf{K}\ \mathbf{G_{23}} = \begin{array}{c} \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 2 & 3 & & \end{array} \\ \begin{pmatrix} 0 & \overrightarrow{0} & \triangle & x & x & x & x & x \\ 0_{\downarrow} & 0 & x & \otimes & 0 & \otimes & \otimes & \otimes \\ \triangle & x & 0 & \otimes & 0 & \otimes & \otimes & \otimes \\ x & \otimes & \otimes & 0 & 0 & 0 & 0 & x \end{pmatrix} \end{array}$$

$$= \begin{pmatrix} 0 & 0 & x & x & x & x & x & x \\ 0 & 0 & x & x & 0 & x & x & x \\ x & x & 0 & x & 0 & \mathbf{x} & x & x \\ x & x & x & 0 & 0 & 0 & 0 & x \end{pmatrix}$$

where $\tilde{\mathbf{G}}_{\mathbf{23}}^{\mathbf{T}}$ is the leading 4×4 principle submatrix of $\mathbf{G}_{ij}$, which is a 8×8 Givens matrix. For simplicity in notation, we assume that $\mathbf{K}$ and $\mathbf{B}, \mathbf{R}$ are over written during the reductions. For instance, for the above reduction,

$$\mathbf{K} \longleftarrow \tilde{\mathbf{G}}_{\mathbf{23}}^{\mathbf{T}}\ \mathbf{K}\ \mathbf{G_{23}}$$

$$\Longleftrightarrow\ \ \mathbf{B} \longleftarrow \tilde{\mathbf{G}}_{\mathbf{23}}^{\mathbf{T}}\ \mathbf{B}\ \tilde{\mathbf{G}}_{\mathbf{23}},\ \ \mathbf{R} \longleftarrow \tilde{\mathbf{G}}_{\mathbf{23}}^{\mathbf{T}}\ \mathbf{R}$$

where matrices $\mathbf{K}$, $\mathbf{B}$, and $\mathbf{R}$ are replaced by the last transformation.

After each iteration in shifting the mass, one entry to the right in submatrix $\mathbf{B}$, a cleanup transformation is required to eliminate the zero contamination in the sub-matrix $\mathbf{R}$. So, after this first iteration, the original submatrix entry r_{32} (or r_{36}) (equal to zero) has been contaminated and now must be cleaned up. This cleanup transformation is $\mathbf{K} \longleftarrow \mathbf{K}\ \mathbf{G_{67}}$.

$$\mathbf{K}\ \mathbf{G_{67}} = \begin{array}{c} \\ 0 \\ 0 \\ x \\ x \end{array} \begin{array}{c} \\ 0 \\ 0 \\ x \\ x \end{array} \begin{array}{c} \\ x \\ x \\ 0 \\ x \end{array} \begin{array}{c} \\ x \\ 0 \\ 0 \\ 0 \end{array} \begin{array}{c} 6 \\ \otimes \\ \otimes \\ \overrightarrow{0} \\ 0 \end{array} \begin{array}{c} 7 \\ \otimes \\ \otimes \\ \triangle \\ 0 \end{array} \begin{array}{c} \\ x \\ x \\ x \\ x \end{array}$$

$$= \begin{pmatrix} 0 & 0 & x & x & x & x & x & x \\ 0 & 0 & x & x & 0 & x & x & x \\ x & x & 0 & x & 0 & 0 & x & x \\ x & x & x & 0 & 0 & 0 & 0 & x \end{pmatrix}$$

Since $\mathbf{G_{67}}$ only affects submatrix $\mathbf{R}$ within $\mathbf{K}$, this means that due to anti-symmetry properties, $\tilde{\mathbf{G}}^{\mathbf{T}}_{\mathbf{67}}$, which is a 4×4 diagonal block of $\mathbf{G_{67}}$, must be saved but does not act on $\mathbf{K}$. The next iteration shifts the mass of $\mathbf{b}_{13}$ to submatrix $\mathbf{B}$'s last row entry, $\mathbf{b}_{14}$.

$$\tilde{\mathbf{G}}^{\mathbf{T}}_{\mathbf{34}}\ \mathbf{K}\ \mathbf{G}_{34} = \begin{pmatrix} 0 & 0 & \overrightarrow{0} & \triangle & x & x & x & x \\ 0 & 0 & \otimes & \otimes & 0 & x & x & x \\ 0_{\downarrow} & \otimes & 0 & x & 0 & 0 & \otimes & \otimes \\ \triangle & \otimes & x & 0 & 0 & 0 & \otimes & \otimes \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 0 & 0 & x & x & x & x & x \\ 0 & 0 & x & x & 0 & x & x & x \\ 0 & x & 0 & x & 0 & 0 & x & x \\ x & x & x & 0 & 0 & 0 & \mathbf{x} & x \end{pmatrix}$$

Again, this shifting of mass to the next entry has contaminated the zero entry in $\mathbf{r}_{43}$ and must be cleaned up.

$$\mathbf{K}\,\mathbf{G}_{78} = \begin{array}{c} \\ \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{cccc} & & & \end{array} \begin{array}{cc} 7 & 8 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & x & x & x & \otimes & \otimes \\ 0 & 0 & x & x & 0 & x & \otimes & \otimes \\ 0 & x & 0 & x & 0 & 0 & \otimes & \otimes \\ x & x & x & 0 & 0 & 0 & \vec{0} & \triangle \end{pmatrix} \end{array}$$

$$= \begin{pmatrix} 0 & 0 & 0 & x & x & x & x & x \\ 0 & 0 & x & x & 0 & x & x & x \\ 0 & x & 0 & x & 0 & 0 & x & x \\ x & x & x & 0 & 0 & 0 & 0 & x \end{pmatrix}$$

Step two is the shifting of submatrix **B**'s first row's last entry, b_{14}, to the first row of the last entry, r_{14} of submatrix **R**. Step two leverages the unique properties of congruence transformations. To illustrate these unique properties, the full matrix **H** is displayed to demonstrate the advantageous and limitations of this step. For notation purposes, **K** is essentially expanded to the full matrix **H** which includes the transformations up to this point as well.

$$\mathbf{G}_{48}^{T}\,\mathbf{H}\,\mathbf{G}_{48} = \begin{array}{c} \\ 4 \\ \\ \\ \\ \\ \\ 8 \end{array} \begin{array}{c} \begin{array}{cccc} & & & \end{array} \begin{array}{cc} 4 & \end{array} \begin{array}{ccc} & & \end{array} \begin{array}{c} 8 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & \vec{0} & x & x & x & \triangle \\ 0 & 0 & x & \otimes & 0 & x & x & \otimes \\ 0 & x & 0 & \otimes & 0 & 0 & x & \otimes \\ 0_{\downarrow} & \otimes & \otimes & 0 & \mathbf{0} & \mathbf{0} & \mathbf{0} & x \\ x & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & \mathbf{0} \\ x & x & 0 & \mathbf{0} & 0 & 0 & 0 & \mathbf{0} \\ x & x & x & \mathbf{0} & 0 & 0 & 0 & \mathbf{0} \\ \triangle & \otimes & \otimes & x & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{pmatrix} \end{array}$$

It is in this step that identifies the power of congruence transformations.

$$
\begin{array}{c} \\ \\ \\ \\ 4 \\ \\ \\ \\ 8 \end{array}
\begin{array}{c}
\begin{array}{cccccccc} & & & 4 & & & & 8 \end{array} \\
\begin{pmatrix}
0 & 0 & 0 & \overrightarrow{0} & x & x & x & \triangle \\
0 & 0 & x & \otimes & 0 & x & x & \otimes \\
0 & x & 0 & \otimes & 0 & 0 & x & \otimes \\
0_{\downarrow} & \otimes & \otimes & \odot & \mathbf{0} & \mathbf{0} & \mathbf{0} & \odot \\
x & 0 & 0 & \mathbf{0} & 0 & 0 & 0 & \mathbf{0} \\
x & x & 0 & \mathbf{0} & 0 & 0 & 0 & \mathbf{0} \\
x & x & x & \mathbf{0} & 0 & 0 & 0 & \mathbf{0} \\
\triangle & \otimes & \otimes & \odot & \mathbf{0} & \mathbf{0} & \mathbf{0} & \odot
\end{pmatrix}
\end{array}
$$

Congruent transformations preserve these values located at $\odot$ which means that no contamination occurs in $h_{2n,2n}$ which in this case is h_{88} . With no contamination occurring to $h_{2n,2n}$, this is a key algorithm property in preserving the **0** submatrix block in **H**. In addition, during step one, after each transformation iteration, the zero contaminations below the diagonal were cleaned up in the block **R**. This cleaning up of all contaminations below the diagonal preserves the safe zones. Safe zones are designated as $\square$ entries. Safe zones are entries which must remain zero before any transformations of information from submatrix **B** to the submatrix **R** can occur.

$$
\begin{array}{c} \\ \\ \\ 4 \\ \\ \\ \\ 8 \end{array}
\begin{array}{c}
\begin{array}{cccccccc} & & & 4 & & & & 8 \end{array} \\
\begin{pmatrix}
0 & 0 & 0 & \overrightarrow{0} & x & x & x & \triangle \\
0 & 0 & x & \otimes & 0 & x & x & \otimes \\
0 & x & 0 & \otimes & 0 & 0 & x & \otimes \\
0_{\downarrow} & \otimes & \otimes & \odot & \square & \square & \square & \odot \\
x & 0 & 0 & \square & 0 & 0 & 0 & \mathbf{0} \\
x & x & 0 & \square & 0 & 0 & 0 & \mathbf{0} \\
x & x & x & \square & 0 & 0 & 0 & \mathbf{0} \\
\triangle & \otimes & \otimes & \odot & \mathbf{0} & \mathbf{0} & \mathbf{0} & \odot
\end{pmatrix}
\end{array}
$$

If the safe zones are allowed to become contaminated, then the **0** submatrix block is no longer preserved. For example, let ⊠ indicate a contaminated safe zone box. When shifting information from submatrix **B** to **R**, the Given's transformation matrix is $\mathbf{G}_{n,2n}$ where the dimensions of both submatrices **B** and **R** are n-by-n. In our current example, this congruent similarity Given's transformation matrix is $\mathbf{G}_{48}$.

$$
\begin{array}{c} \\ \\ \\ 4 \\ \\ \\ \\ 8 \end{array}
\begin{array}{c}
\begin{array}{cccccccc} & & & 4 & & & & 8 \end{array} \\
\begin{pmatrix}
0 & 0 & 0 & \overrightarrow{0} & x & x & x & \triangle \\
0 & 0 & x & \otimes & 0 & x & x & \otimes \\
0 & x & 0 & \otimes & 0 & 0 & x & \otimes \\
0_{\downarrow} & \otimes & \otimes & \odot & \square & \square & \boxtimes & \odot \\
x & 0 & 0 & \square & 0 & 0 & 0 & \mathbf{0} \\
x & x & 0 & \square & 0 & 0 & 0 & \mathbf{0} \\
x & x & x & \boxtimes & 0 & 0 & 0 & \otimes \\
\triangle & \otimes & \otimes & \odot & \mathbf{0} & \mathbf{0} & \otimes & \odot
\end{pmatrix}
\end{array}
$$

When applying the transformation to shift information from submatrix **B** to **R**, the

contaminated safe zone boxes causes contamination to spread into the submatrix **0**. Because congruent transformations preserve the $\odot$ entries, it is the only transformation which can shift information from **B** to **R** without contaminating the submatrix **0**. Therefore, congruent transformations are the key to this algorithm and makes it imperative that the safe zone blocks remain zero entries.

Step three is adjusting submatrix **R**'s first row in order to repeat stage one through three again on the next subsequent row. In addition, stage three conveniently sets the first row of matrix **R** toward an upper bidiagonal form. In stage three's first iteration, the information mass in $r_{1,n}$ is transferred to $r_{1,n-1}$. In this case, it would be r_{14} to r_{13}.

$$\mathbf{K} \longleftarrow \mathbf{K}\ \mathbf{G}_{78} = \begin{array}{c} \\ \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{cccccccc} & & & & & & 7 & 8 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & x & x & \triangle & \overleftarrow{0} \\ 0 & 0 & x & x & 0 & x & \otimes & \otimes \\ 0 & x & 0 & x & 0 & 0 & \otimes & \otimes \\ 0 & x & x & 0 & 0 & 0 & \boxtimes & \otimes \end{pmatrix} \end{array}.$$

It is evident that this iteration contaminated one of the safe zone blocks. A quick transformation is conducted to clean up the below diagonal contaminations and in this case a contaminated safe zone block.

$$\mathbf{K} \longleftarrow \tilde{\mathbf{G}}_{\mathbf{34}}^{\mathbf{T}}\ \mathbf{K}\ \mathbf{G}_{\mathbf{34}} = \begin{array}{c} \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccccccc} & & & & 3 & 4 & & \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & x & x & x & 0 \\ 0 & 0 & \otimes & \otimes & 0 & x & x & x \\ 0 & \otimes & 0 & x & 0 & 0 & \triangle & \otimes \\ 0 & \otimes & x & 0 & 0 & 0 & 0^{\uparrow} & \otimes \end{pmatrix} \end{array}$$

Continuing to shift the first row of matrix **R** toward an upper bidiagonal form, the next iteration is

$$\mathbf{KG}_{67} = \begin{array}{c} \\ \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{cccc} & & & & & & 6 & 7 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & x & \triangle & \overleftarrow{0} & 0 \\ 0 & 0 & x & x & 0 & \otimes & \otimes & x \\ 0 & x & 0 & x & 0 & \otimes & \otimes & x \\ 0 & x & x & 0 & 0 & 0 & 0 & x \end{pmatrix} \end{array}$$

$$= \begin{pmatrix} 0 & 0 & 0 & 0 & x & x & 0 & 0 \\ 0 & 0 & x & x & 0 & x & x & x \\ 0 & x & 0 & x & 0 & \mathbf{x} & x & x \\ 0 & x & x & 0 & 0 & 0 & 0 & x \end{pmatrix}$$

Likewise, for each transformation of matrix **K**, assume that the replacement operation is conducted for the remainder of this study. Again, due to **K** $\mathbf{G}_{67}$, a transformation is required to clean up its contamination located below the diagonal of the submatrix **R**.

$$\tilde{\mathbf{G}}_{23}^{T}\ \mathbf{K}\ \mathbf{G}_{23} = \begin{array}{c} \\ 2 \\ 3 \\ \\ \end{array} \begin{array}{c} \begin{array}{cccc} & 2 & 3 & & & & & \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & x & x & 0 & 0 \\ 0 & 0 & x & x & 0 & \triangle & \otimes & \otimes \\ 0 & x & 0 & x & 0 & 0^{\uparrow} & \otimes & \otimes \\ 0 & \otimes & \otimes & 0 & 0 & 0 & 0 & x \end{pmatrix} \end{array}$$

$$= \begin{pmatrix} 0 & 0 & 0 & 0 & x & x & 0 & 0 \\ 0 & 0 & x & x & 0 & x & x & x \\ 0 & x & 0 & x & 0 & 0 & x & x \\ 0 & x & x & 0 & 0 & 0 & 0 & x \end{pmatrix}$$

Step three is now complete for the first row. Essentially, step three's iterative transformations shifted all information from $r_{1,n} \rightarrow r_{1,n-1} \rightarrow \cdots \rightarrow r_{1,2}$ with appropriate clean up transformations required for each iteration.

This three step process on shifting the mass from submatrix **B** to submatrix **R** for the first row is summarized:

$$\mathbf{Q_{12}^T Q_{11}^T\ H\ Q_{11} Q_{12}} = \begin{pmatrix} 0 & 0 & 0 & 0 & x & x & 0 & 0 \\ 0 & 0 & x & x & 0 & x & x & x \\ 0 & x & 0 & x & 0 & 0 & x & x \\ 0 & x & x & 0 & 0 & 0 & 0 & x \\ x & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ x & x & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & x & x & 0 & 0 & 0 & 0 & 0 \\ 0 & x & x & x & 0 & 0 & 0 & 0 \end{pmatrix} \tag{4.10}$$

where $\mathbf{Q_{11}} = \mathbf{G_{23}\ G_{67}\ G_{34}\ G_{78}\ G_{48}}$, $\mathbf{Q_{12}} = \mathbf{G_{78}\ G_{34}\ G_{67}\ G_{23}}$.

$\mathbf{Q_{11}}$ and its transpose are all the Givens transformations which reduce matrix **B** and any cleanup operations associated to that operation. $\mathbf{Q_{12}}$ and its transpose are relate to the reduction of matrix **R** to an upper bidiagonal and the associated cleanup operations. With the first row completed, this process is then repeated for each row until the submatrix **B**'s entire mass has been shifted to **R** and **R** is now

equal to matrix **U** an upper bidiagonal matrix. A simpler notation will quickly illustrate how the remaining two rows are transformed by the 3-step process:

$$\tilde{\mathbf{G}}_{\mathbf{34}}^{\mathbf{T}}\mathbf{K}\mathbf{G}_{34} = \begin{array}{c} \\ 3 \\ 4 \end{array}\!\!\begin{array}{c} \begin{array}{cccc} & & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \overrightarrow{0} & \triangle & 0 & x & x & x \\ 0 & 0_{\downarrow} & 0 & x & 0 & 0 & \otimes & \otimes \\ 0 & \triangle & x & 0 & 0 & 0 & \mathbf{x} & \otimes \end{pmatrix} \end{array} \rightarrow$$

$$\begin{array}{c} \begin{array}{cccc} & & 7 & 8 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & x & x & 0 & 0 \\ 0 & 0 & 0 & x & 0 & x & \otimes & \otimes \\ 0 & 0 & 0 & x & 0 & 0 & \otimes & \otimes \\ 0 & x & x & 0 & 0 & 0 & \overrightarrow{0} & \triangle \end{pmatrix} \end{array} \rightarrow \begin{array}{c} \\ 4 \end{array}\!\!\begin{array}{c} \begin{array}{cccc} & & 4 & \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & x & x & 0 & 0 \\ 0 & 0 & 0 & \overrightarrow{0} & 0 & x & x & \triangle \\ 0 & 0 & 0 & \otimes & 0 & 0 & x & \otimes \\ 0 & 0_{\downarrow} & x & \odot & 0 & 0 & 0 & \odot \end{pmatrix} \end{array} \begin{array}{c} \\ \\ 8 \end{array}$$

$$\rightarrow \begin{array}{c} \begin{array}{cccc} & & 7 & 8 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & x & x & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & x & \triangle & \overleftarrow{0} \\ 0 & 0 & 0 & x & 0 & 0 & \otimes & \otimes \\ 0 & 0 & x & 0 & 0 & 0 & \mathbf{x} & \otimes \end{pmatrix} \end{array} \rightarrow \begin{array}{c} \\ 3 \\ 4 \end{array}\!\!\begin{array}{c} \begin{array}{cccc} & & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & x & x & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & x & x & 0 \\ 0 & 0 & 0 & x & 0 & 0 & \triangle & \otimes \\ 0 & 0 & x & 0 & 0 & 0 & 0^{\uparrow} & \otimes \end{pmatrix} \end{array}$$

$$\rightarrow \begin{array}{c} \\ 4 \end{array}\!\!\begin{array}{c} \begin{array}{cccc} & & 4 & \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & x & x & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & x & x & 0 \\ 0 & 0 & 0 & \overrightarrow{0} & 0 & 0 & x & \triangle \\ 0 & 0 & 0_{\downarrow} & 0 & 0 & 0 & 0 & x \end{pmatrix} \end{array} \begin{array}{c} \\ \\ 8 \end{array} \rightarrow \begin{pmatrix} 0 & 0 & 0 & 0 & x & x & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & x & x & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & x \end{pmatrix}.$$

Now, when expanding the above **K** matrix into -**H**'s full factorized form, it is denoted as matrix **X**. This matrix is not quite in the real Schur form. The last stage in this factorization is to transform this expanded matrix **X** into the real Schur form.

$$\mathbf{X} = \begin{pmatrix} 0 & 0 & 0 & 0 & x & x & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & x & x & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & x \\ -x & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -x & -x & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -x & -x & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -x & -x & 0 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & \mathbf{U} \\ -\mathbf{U^T} & 0 \end{pmatrix}.$$

The two non zero submatrices, **U**, which is upper bidiagonal, and $\mathbf{U^T}$, of matrix **X** are then diagonalized using an SVD similarity-congruent transformation.

$$\mathbf{U} = \tilde{\mathbf{U}}\mathbf{\Sigma}\mathbf{V^T} \implies -\mathbf{U^T} = -\mathbf{V}\mathbf{\Sigma}\tilde{\mathbf{U}}^{\mathbf{T}} \quad \text{then}$$

$$\begin{pmatrix} \mathbf{0} & \mathbf{U} \\ -\mathbf{U^T} & \mathbf{0} \end{pmatrix} = \begin{pmatrix} \mathbf{0} & \tilde{\mathbf{U}}\mathbf{\Sigma}\mathbf{V^T} \\ -\mathbf{V}\mathbf{\Sigma}\tilde{\mathbf{U}}^{\mathbf{T}} & 0 \end{pmatrix} =$$

$$\begin{pmatrix} \tilde{\mathbf{U}} & \mathbf{0} \\ \mathbf{0} & \mathbf{V} \end{pmatrix} \begin{pmatrix} \mathbf{0} & \mathbf{\Sigma} \\ -\mathbf{\Sigma} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \tilde{\mathbf{U}}^{\mathbf{T}} & \mathbf{0} \\ \mathbf{0} & \mathbf{V^T} \end{pmatrix} = \mathbf{Q}\ \tilde{\mathbf{T}}\ \mathbf{Q^T}$$

where $\mathbf{Q} = \begin{pmatrix} \tilde{\mathbf{U}} & \mathbf{0} \\ \mathbf{0} & \mathbf{V} \end{pmatrix}$ is orthogonal and

$$\tilde{\mathbf{T}} = \begin{pmatrix} \mathbf{0} & \mathbf{\Sigma} \\ -\mathbf{\Sigma} & \mathbf{0} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & 0 & x & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & x & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & x & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & x \\ -x & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -x & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -x & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -x & 0 & 0 & 0 & 0 \end{pmatrix} \tag{4.11}$$

Matrix -**H** is now in its final decomposition form which is the real Schur form.

In this diagonalization process, only one method was presented for step 1 and 2. No other method appears to be faster nor able to preserve the (2,2) zero block. As an example, lets apply the Householder reflector.

Householder, $\mathbf{Q}_k$, when applied to a vector can collapse the mass elements to fewer mass elements.

$$\mathbf{Q}_k \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_k \\ x_{k+1} \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 0 \\ x_{k+1} \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix}$$

Note, for an $m \times m$ $\mathbf{H}$ matrix where $m = 2n$, then the modified Householder transformation matrix will take the following form:

$$\text{Householder } \mathbf{Q}(\mathbf{k}) = \begin{pmatrix} \mathbf{Q_k} & 0 \\ 0 & \mathbf{I} \end{pmatrix}$$

If we apply the Householder to reduce matrix $\mathbf{B}$ which is block (1,1), we get the following:

$$\mathbf{Q}_4^T \begin{pmatrix} 0 & x & x & x & x & x & 0 & 0 \\ x & 0 & x & x & 0 & x & x & x \\ x & x & 0 & x & 0 & 0 & x & x \\ x & x & x & 0 & 0 & 0 & 0 & x \end{pmatrix} \mathbf{Q}_4 =$$

$$\begin{pmatrix} 0 & 0 & 0 & x & \otimes & \otimes & \otimes & \otimes \\ 0 & 0 & x & x & \otimes & \otimes & \otimes & \otimes \\ 0 & x & 0 & x & \otimes & \otimes & \otimes & \otimes \\ x & x & x & 0 & \otimes & \otimes & \otimes & \otimes \end{pmatrix}$$

So, if we apply Householder iteratively, we can reduce matrix **B** to the following form

$$\begin{pmatrix} 0 & 0 & 0 & x & \otimes & \otimes & \otimes & \otimes \\ 0 & 0 & 0 & x & \otimes & \otimes & \otimes & \otimes \\ 0 & 0 & 0 & x & \otimes & \otimes & \otimes & \otimes \\ x & x & x & 0 & \otimes & \otimes & \otimes & \otimes \end{pmatrix}$$

However, when we attempt to transfer the last column in matrix **B** to the transformed matrix block (1,2), the (2,2) zero block is then contaminated. Since the safe zone blocks are contaminated, any mass sent from block (1,1) to (1,2) will contaminate the (2,2) zero block. Before we can shift the mass over, the (1,2) matrix block must again be reduced to an upper triangular matrix. This can be done but the clean up cost is larger than The Method. Also, when applying the Householder iteratively, the $\mathbf{Q_k}$ is acting on all the contamination in (1,2) upper triangular block. Thus, this is just one example why the Givens transformation along with the The Method appears to be the only choice.

So, The Method appears to be the cheapest route to the upper bidiagonal form and then the SVD transformations is used to get the real Schur form of **H**.

Chapter 5

Analyze The Methods Numerical Approach

5.1 Flop Analysis of Algorithm

The challenge in flop analysis is that all algorithms which determine five or more eigenvalues must incorporate an infinite iterative process to sift out the eigenvalues. With a predefined tolerance level established, the infinite iteration is stopped when the values of the estimated eigenvalues are close enough to the exact eigenvalues. In The Method's case, the iterative process occurs at the Singular Value Decomposition (SVD) step. For Matlab's eig(), the iterative process is in the iterative QR process. Consequently, in comparing whether The Method's flops are less than or greater than Matlab's eig() QR process, the flop comparative analysis begins with the matrix structure **H** and ends after the first iteration is complete.

In The Method, the operation used prior to implementing the SVD iterations is the Givens transformation matrix. Each time a 2×2 Givens transformation matrix

is applied to a 2-dimensional vector, six flops are processed.

The Method's flop analysis is broken down in to four steps. **K** is an n-by-$2n$ matrix where submatrix **B** and **R** are n-by-n in dimensions.

(1) Reduction of matrix B to column nth and theassociated contamination and c

(2) Transfer of matrix B's nth column to matrix R's nth column

(3) Reduction of matrix R to bidiagonal and effects on matrix B

(4) SVD cost to reduce bidiagonal matrix R

(5) Theorem

(1) Reduction of matrix B to the nth column and the associated contamination and cleanup of R

Since **B** is skew symmetric, Given's congruence transformations do not change the diagonal blocks

$$\begin{array}{c} \\ i \\ j \\ \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{cccccc} i & j & & & & \end{array} \\ \begin{pmatrix} 0 & x & x & x & x & x \\ x & \odot & \odot & x & x & x \\ x & \odot & \odot & x & x & x \\ x & x & x & 0 & x & x \\ x & x & x & x & 0 & x \\ x & x & x & x & x & 0 \end{pmatrix} \end{array}$$

Although The Method literally does not reduce matrix $\mathbf{B}$ to the nth column, for the purposes of flop analysis this form is essentially correct. Since $\mathbf{B}$ is skew-symmetric, we only have to calculate the effects of the right and left Givens matrix on entries either below or above the diagonal. All changes above the diagonal are the equivalent skew-symmetric counterparts for the entries located below diagonal of submatrix $\mathbf{B}$. In this study, the entries above the diagonal are chosen for the analysis. From the reduction of $\mathbf{B}$, submatrix $\mathbf{R}$'s below diagonal entries are contaminated and are subsequently eliminated. A matrix $\mathbf{K} \in \mathbf{R}^{\mathbf{6\times 2*6}}$ is used to illustrate the flop costs where $\mathbf{K} = [\mathbf{BR}]$.

Clearing the first row of $\mathbf{B}$ and contamination and cleanup of $\mathbf{R}$ will cost $2s_1 * (s_1 - 2)$ where $s_1 = n = 6$.

$$
\begin{array}{c} \\ 2 \\ 3 \end{array}
\begin{array}{c}
\begin{array}{cccccc|cccccc} 2 & 3 & & & & & & & & & & \end{array} \\
\left(\begin{array}{cc|cccc|cccccc}
0 & | & | & \mathbf{x} & \mathbf{x} & \mathbf{x} & x & x & x & x & x & x \\
x & 0 & x & - & - & - & 0 & - & - & - & - & - \\
x & x & 0 & - & - & - & 0 & - & - & - & - & - \\
x & x & x & x & x & x & 0 & 0 & 0 & x & x & x \\
x & x & x & x & 0 & x & 0 & 0 & 0 & 0 & x & x \\
x & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & x
\end{array}\right)
\end{array}
\mapsto
$$

$$
\begin{array}{c}
\begin{array}{cccccc|cccccc} & & & & & & 8 & 9 & & & & \end{array} \\
\left(\begin{array}{cc|cccc|cc|ccc}
0 & 0 & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & x & | & | & x & x & x \\
x & 0 & x & x & x & x & 0 & | & | & x & x & x \\
x & x & 0 & x & x & x & 0 & | & | & x & x & x \\
x & x & x & 0 & x & x & 0 & 0 & 0 & x & x & x \\
x & x & x & x & 0 & x & 0 & 0 & 0 & 0 & x & x \\
x & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & x
\end{array}\right)
\end{array}
$$

where the cost is 6*12 where 12 is the number of givens rotations acting on some (x_1, x_2) vectors, and 6 is the flop cost for each Givens rotation. One givens rotation on a vector is denoted as $\begin{pmatrix} | & | \end{pmatrix}$ or $\begin{pmatrix} - \\ \\ - \end{pmatrix}$, and matrix $\mathbf{S_1} \in \mathbf{R^{6\times 6}}$

$$\cdot \begin{pmatrix} & & x & x & x & x & x & x \\ & & 0 & x & x & x & x & x \\ & S_1 & 0 & 0 & x & x & x & x \\ & & 0 & 0 & 0 & x & x & x \\ & & 0 & 0 & 0 & 0 & x & x \\ & & 0 & 0 & 0 & 0 & 0 & x \end{pmatrix}$$

Continuing with this process on the first row,

$$\begin{array}{c} \\ \\ 3 \\ 4 \\ \\ \\ \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{cccccc} & & 3 & 4 & & \end{array} \\ \begin{pmatrix} 0 & 0 & | & | & \mathbf{x} & \mathbf{x} & x & x & x & x & x & x \\ x & 0 & | & | & x & x & 0 & x & x & x & x & x \\ x & x & 0 & x & - & - & 0 & 0 & - & - & - & - \\ x & x & x & 0 & - & - & 0 & 0 & - & - & - & - \\ x & x & x & x & 0 & x & 0 & 0 & 0 & 0 & x & x \\ x & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & x \end{pmatrix} \end{array} \mapsto$$

$$\begin{array}{c} \begin{array}{cccccc} & & & & & & & & 9 & 10 & & \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & \mathbf{x} & \mathbf{x} & \mathbf{x} & x & x & | & | & x & x \\ x & 0 & x & x & x & x & 0 & x & | & | & x & x \\ x & x & 0 & x & x & x & 0 & 0 & | & | & x & x \\ x & x & x & 0 & x & x & 0 & 0 & | & | & x & x \\ x & x & x & x & 0 & x & 0 & 0 & 0 & 0 & x & x \\ x & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & x \end{pmatrix} \end{array}$$

where the cost is 6*12; 12 is the number of givens rotations; and 6 is the flop cost for each Givens rotation. Again, continuing with this process on the first row,

$$
\begin{array}{c} \\ \\ \\ 4 \\ 5 \\ \\ \end{array}
\begin{array}{c}
\begin{array}{cccccc} & & & 4 & 5 & \end{array} \\
\left(\begin{array}{cccccc|cccccc}
0 & 0 & 0 & | & | & \mathbf{x} & x & x & x & x & x & x \\
x & 0 & x & | & | & x & 0 & x & x & x & x & x \\
x & x & 0 & | & | & x & 0 & 0 & x & x & x & x \\
x & x & x & 0 & x & - & 0 & 0 & 0 & - & - & - \\
x & x & x & x & 0 & - & 0 & 0 & 0 & - & - & - \\
x & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & x
\end{array}\right)
\end{array}
\mapsto
$$

$$
\begin{array}{c}
\begin{array}{cccccc} & & & & & 10 \quad 11 \end{array} \\
\left(\begin{array}{cccccc|cccccc}
0 & 0 & 0 & 0 & \mathbf{x} & \mathbf{x} & x & x & x & | & | & x \\
x & 0 & x & x & x & x & 0 & x & x & | & | & x \\
x & x & 0 & x & x & x & 0 & 0 & x & | & | & x \\
x & x & x & 0 & x & x & 0 & 0 & 0 & | & | & x \\
x & x & x & x & 0 & x & 0 & 0 & 0 & | & | & x \\
x & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & x
\end{array}\right)
\end{array}
$$

where the cost is 6*12 for the same reasons.

$$
\begin{array}{c} \\ \\ \\ \\ 5 \\ 6 \\ \end{array}
\begin{array}{c}
\begin{array}{cccccc} & & & & 5 & 6 \end{array} \\
\left(\begin{array}{cccccc|cccccc}
0 & 0 & 0 & 0 & | & | & x & x & x & x & x & x \\
x & 0 & x & x & | & | & 0 & x & x & x & x & x \\
x & x & 0 & x & | & | & 0 & 0 & x & x & x & x \\
x & x & x & 0 & | & | & 0 & 0 & 0 & x & x & x \\
x & x & x & x & 0 & x & 0 & 0 & 0 & 0 & - & - \\
x & x & x & x & x & 0 & 0 & 0 & 0 & 0 & - & -
\end{array}\right)
\end{array}
\mapsto
$$

$$\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array} \begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array} \begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array} \begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array} \begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array} \begin{array}{c} 11 \\ | \\ | \\ | \\ | \\ | \\ | \end{array} \begin{array}{c} 12 \\ | \\ | \\ | \\ | \\ | \\ | \end{array}$$

$$\begin{pmatrix} 0 & 0 & 0 & \mathbf{x} & \mathbf{x} & \mathbf{x} & x & x & x & x \\ x & 0 & x & x & x & x & 0 & x & x & x \\ x & x & 0 & x & x & x & 0 & 0 & x & x \\ x & x & x & 0 & x & x & 0 & 0 & x & x \\ x & x & x & x & 0 & x & 0 & 0 & 0 & 0 \\ x & x & x & x & x & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

where the cost is 6*12 for the same reasons. The total cost for reduction and cleanup operations of matrix **K**'s first row, the total cost was $4 * (6 * 12)$. This total can be generically rewritten as $(s_1 - 2) * 6 * (2 * s_1)$.

When reducing the second row of matrix **K**, $\mathbf{S_2} \in \mathbf{R^{s_2 \times s_2}} = \mathbf{R^{5 \times 5}}$

$$\cdot \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & x & x & 0 & 0 & 0 & 0 \\ 0 & & & & & 0 & x & x & x & x & x \\ 0 & & & & & 0 & 0 & x & x & x & x \\ 0 & & S_2 & & & 0 & 0 & 0 & x & x & x \\ 0 & & & & & 0 & 0 & 0 & 0 & x & x \\ 0 & & & & & 0 & 0 & 0 & 0 & 0 & x \end{pmatrix}$$

$$\begin{array}{c} \\ \\ 3 \\ 4 \\ \\ \\ \end{array} \begin{array}{c} 3 \\ 4 \\ | \\ | \\ | \\ | \\ | \\ | \end{array} \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & x & x & 0 & 0 & 0 & 0 \\ 0 & 0 & | & | & x & x & 0 & x & x & x & x & x \\ 0 & x & 0 & x & - & - & 0 & 0 & - & - & - & - \\ 0 & x & x & 0 & - & - & 0 & 0 & - & - & - & - \\ 0 & x & x & x & 0 & x & 0 & 0 & 0 & 0 & x & x \\ 0 & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & x \end{pmatrix} \quad \mapsto$$

$$\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} 9 \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} 10 \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & x & x & | & | & x & x \\ 0 & 0 & x & x & x & x & 0 & x & | & | & x & x \\ 0 & x & 0 & x & x & x & 0 & 0 & | & | & x & x \\ 0 & x & x & 0 & x & x & 0 & 0 & | & | & x & x \\ 0 & x & x & x & 0 & x & 0 & 0 & 0 & 0 & x & x \\ 0 & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & x \end{pmatrix}$$

where the cost is 6*10; 10 is the number of givens rotations; and 6 is the flop cost for each Givens rotation. Again, continuing with this process on the second row,

$$\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} 4 \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} 5 \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}$$

$$\begin{array}{c} \\ 4 \\ 5 \end{array}\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & x & x & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & | & | & x & 0 & x & x & x & x & x \\ 0 & x & 0 & | & | & x & 0 & 0 & x & x & x & x \\ 0 & x & x & 0 & x & - & 0 & 0 & 0 & - & - & - \\ 0 & x & x & x & 0 & - & 0 & 0 & 0 & - & - & - \\ 0 & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & x \end{pmatrix} \quad \mapsto$$

$$\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} 10 \\ \\ \\ \\ \\ \\ \end{array}\begin{array}{c} 11 \\ \\ \\ \\ \\ \\ \end{array}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & x & x & x & | & | & x \\ 0 & 0 & x & x & x & x & 0 & x & x & | & | & x \\ 0 & x & 0 & x & x & x & 0 & 0 & x & | & | & x \\ 0 & x & x & 0 & x & x & 0 & 0 & 0 & | & | & x \\ 0 & x & x & x & 0 & x & 0 & 0 & 0 & | & | & x \\ 0 & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & x \end{pmatrix}$$

where the cost is 6*10 for the same reasons. This process will continue with the following results second row reduction. The total cost for reduction and cleanup operations of matrix **K**'s second row, the total cost was $3 * (6 * 10)$. This total can be generically rewritten as $(s_2 - 2) * 6 * (2 * s_2)$.

Thus, the iteration for reduction and cleanup of matrix **K** follows these generic steps:

Row 1: $(s_1 - 2) * 6 * (2 * s_1)$ **flops**

Row 2: $(s_2 - 2) * 6 * (2 * s_2)$ **flops**

Row 3: $(s_3 - 2) * 6 * (2 * s_3)$ **flops**

$\vdots$

Row k: $(s_k - 2) * 6 * (2 * s_k)$ **flops**

Since when the dim of **S** is 2, the reduction is over for this step. This means that when summing these iterations, k begins at 3. The total flop cost for step 1 is:

$$\sum_{k=3}^{n}(k-2) * 6 * (2 * k)$$

Using the the following two identities

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \Longrightarrow \sum_{k=1}^{n-2} k^2 = \frac{(n-1)(n-2)(2n-3)}{6}$$

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} \Longrightarrow \sum_{k=1}^{n-2} k = \frac{(n-1)(n-2)}{2}$$

Step 1's final cost for the reduction of matrix **B** to the nth column and the cleanup of **R** is

$$\mathbf{12 + 14n - 18n^2 + 4n^3} \approx \mathbf{-18n^2 + 4n^3} \text{ flops}$$

(2) Transfer of matrix B's nth column to matrix R's nth column

Step 2 basically transfers the mass of the nth column in submatrix **B** to the nth column of submatrix **R**.

$$\begin{pmatrix} 0 & 0 & 0 & 0 & x & x & x & x & x & x \\ 0 & 0 & 0 & 0 & x & 0 & x & x & x & x \\ 0 & 0 & 0 & 0 & x & 0 & 0 & x & x & x \\ 0 & 0 & 0 & 0 & x & 0 & 0 & 0 & x & x \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x \end{pmatrix} \mapsto \begin{array}{c} \\ \\ \\ \\ \end{array} \begin{array}{c} \\ \\ \\ \\ \end{array} \begin{array}{c} i \\ \\ \\ \\ \end{array} \begin{array}{c} \\ \\ \\ \\ \end{array} \begin{array}{c} j \\ \\ \\ \\ \end{array} \begin{pmatrix} 0 & 0 & 0 & 0 & | & x & x & x & x & | \\ 0 & 0 & 0 & 0 & | & 0 & x & x & x & | \\ 0 & 0 & 0 & 0 & | & 0 & 0 & x & x & | \\ 0 & 0 & 0 & 0 & | & 0 & 0 & 0 & x & | \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x \end{pmatrix}$$

This transformation occurs $n-1$ times. The total cost for step 2 is $6(n-1)^2$ $= 6n^2 - 12n + 6 \approx 6n^2$

(3) Reduction of matrix R to bidiagonal and effects on matrix B

When reducing each row to a bidiagonal structure on submatrix **R**, the cost is both in row reduction **R** and contamination costs on submatrix **B**.

First row reduction of submatrix **R** and contamination of **B** is

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} & | & | \\ 0 & 0 & x & x & | & | & 0 & x & x & x & | & | \\ 0 & x & 0 & x & | & | & 0 & 0 & x & x & | & | \\ 0 & x & x & 0 & | & | & 0 & 0 & 0 & x & | & | \\ 0 & x & x & x & 0 & x & 0 & 0 & 0 & 0 & + & + \\ 0 & x & x & x & x & 0 & 0 & 0 & 0 & 0 & + & + \end{pmatrix} \mapsto \mathbf{6(2n\text{-}1)}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{x} & \mathbf{x} & \mathbf{x} & | & | & 0 \\ 0 & 0 & x & | & | & x & 0 & x & x & | & | & x \\ 0 & x & 0 & | & | & x & 0 & 0 & x & | & | & x \\ 0 & x & x & 0 & x & - & 0 & 0 & 0 & + & + & - \\ 0 & x & x & x & 0 & - & 0 & 0 & 0 & + & + & - \\ 0 & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & x \end{pmatrix} \mapsto \mathbf{6(2n\text{-}1)}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{x} & \mathbf{x} & | & | & 0 & 0 \\ 0 & 0 & | & | & x & x & 0 & x & | & | & x & x \\ 0 & x & 0 & x & - & - & 0 & 0 & + & + & - & - \\ 0 & x & x & 0 & - & - & 0 & 0 & + & + & - & - \\ 0 & x & x & x & 0 & x & 0 & 0 & 0 & 0 & x & x \\ 0 & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & x \end{pmatrix} \mapsto \mathbf{6(2n\text{-}1)}$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{x} & | & | & 0 & 0 & 0 \\ 0 & 0 & x & - & - & - & 0 & + & + & - & - & - \\ 0 & x & 0 & - & - & - & 0 & + & + & - & - & - \\ 0 & x & x & 0 & x & x & 0 & 0 & 0 & x & x & x \\ 0 & x & x & x & 0 & x & 0 & 0 & 0 & 0 & x & x \\ 0 & x & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & x \end{pmatrix} \mapsto \mathbf{6(2n\text{-}1)}$$

Therefore, the total first row reduction and associated contamination cost is:

$$6(2n-1)+6(2n-1)+6(2n-1)+6(2n-1) = (n-2)6(2n-1) = (n-2)6(2n+1-R_1)$$

The reduction of the second row, R_2, and the associated contamination cost is as follows:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & x & x & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{x} & \mathbf{x} & \mathbf{x} & | & | \\ 0 & 0 & 0 & x & | & | & 0 & 0 & x & x & | & | \\ 0 & 0 & x & 0 & | & | & 0 & 0 & 0 & x & | & | \\ 0 & 0 & x & - & 0 & x & 0 & 0 & 0 & 0 & + & + \\ 0 & 0 & x & x & x & 0 & 0 & 0 & 0 & 0 & + & + \end{pmatrix} \mapsto 6(2n+1-2R_2)$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & x & x & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{x} & \mathbf{x} & | & | & 0 \\ 0 & 0 & 0 & | & | & x & 0 & 0 & x & | & | & x \\ 0 & 0 & x & 0 & x & - & 0 & 0 & 0 & + & + & - \\ 0 & 0 & x & x & 0 & - & 0 & 0 & 0 & + & + & - \\ 0 & 0 & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & x \end{pmatrix} \mapsto 6(2n+1-2R_2)$$

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & x & x & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{x} & | & | & 0 & 0 \\ 0 & 0 & 0 & x & x & x & 0 & 0 & + & + & - & - \\ 0 & 0 & x & 0 & x & x & 0 & 0 & + & + & - & - \\ 0 & 0 & x & x & 0 & x & 0 & 0 & 0 & 0 & x & x \\ 0 & 0 & x & x & x & 0 & 0 & 0 & 0 & 0 & 0 & x \end{pmatrix} \mapsto 6(2n+1-2R_2)$$

The total first row reduction and associated contamination cost is:

$$6(2n+1-2R_2) + 6(2n+1-2R_2) + 6(2n+1-2R_2) = (n-3)6(2n+1-2R_2)$$

The iteration for reduction follows these steps:

Row 1: $(n-2)6(2n+1-2R_1)$

Row 2: $(n-3)6(2n+1-2R_2)$

Row 3: $(n-4)6(2n+1-2R_3)$

$\vdots$

Row k: $(n-1-k)6(2n+1-2k)$

The total cost is reduction of $\mathbf{R}$ to bidiagonal and the associated cross contamination of sub-matrix $\mathbf{B}$ is

$\sum_{\mathbf{k=1}}^{\mathbf{n-2}}\mathbf{(n-1-k)6(2n+1-2k)}$

This expression equals to $\mathbf{6-n-9n^2+4n^3} \approx \mathbf{9n^2+4n^3}$ flops for step 3

Therefore, when summing up all three costs steps in achieving a bidiagonal form , the entire flop cost is then equal to $\mathbf{8n^3-21n^2}$

(4) SVD cost to reduce bidiagonal matrix R

For one bidiagonal iteration, the SVD flop cost using Golub-Kahan SVD algorithm is **32n** flops [9].

(5) Theorem The Method's reduction cost of matrix $\mathbf{H}$ to the Schur form is $\frac{8n^3}{3}$ faster than the Symmetric QR algorithm.

Proof:

- The Method's total reduction cost to bidiagonal is $\mathbf{8n^3-21n^2}$.
- Golub-Kahan SVD cost to reduce from bidiagonal to diagonal is $\mathbf{32n^2}$[9].
- Total cost is $\mathbf{8n^3+11n^2} \approx \mathbf{8n^3}$.

- QR algorithm cost for diagonalizing $m \times m$ matrix where $m = 2n$, is $\frac{\mathbf{4m^3}}{\mathbf{3}} = \frac{\mathbf{4(2n)^3}}{\mathbf{3}} = \frac{\mathbf{32n^3}}{\mathbf{3}}$ flops[9].
- Therefore, The Method is approximately $\frac{\mathbf{8n^3}}{\mathbf{3}}$ flops faster than the standard QR algorithm.
- QED

Since this study is comparing the cost of QR algorithm against The Methods algorithm, The Method is theoretically faster by $\frac{\mathbf{8n^3}}{\mathbf{3}}$ flops. Although not conducted, flop analysis for creating the transformation matrix **Q** would be required for a full comparison.

5.2 Numerically Test Eigenvalue Bounds

Since there is no numerical procedure that can extract exact eigenvalues from a matrix with dimensions greater than 5×5, this study will consider Matlab's eigenvalue outputs from the function eig() as pseudo-exact. The estimated eigenvalues produced from The Method's application on matrix **H** is then compared to Matlab's pseudo-exact eigenvalues. These comparisons are then displayed in several tables. For clarity purpose, exact eigenvalues refers to the eigenvalues determined from the Matlab's eig() function and the estimated eigenvalues are from The Method()'s process.

Test findings: Although not statistically definitive nor fully comprehensive, the numerical test indicates that The Method's eigenvalue error bounds are well behaved. The error bounds C(m) are relatively small in the order of one magnitude or less of matrix **H**'s dimensional size m.

The numerical experiment was structured as follows:

Experimental objective: Determine the behavior of the absolute error between Matlab's eig() function's pseudo-exact eigenvalues and The Method()'s eigenvalues of a given matrix **H** structure.

Equipment:

computer manufacture: Gateway; model: GT5408; Processor: Intel(R)Core(Tm)2 CPU 4300 @ 1.80GHz 1.80GHz; Ram Memory: 1022 MB; System Type: 32-bit Operating System; Rating: 4.5; Machine Precision: 1.1102×10^{16}.

software Matlab & Simulink student version, Version 7.4.0286(R2007a) by The MathWorks

Setup: Since matrix **H** is skew symmetric, then its eigenvalues come in conjugate pairs of plus or minus pure imaginary eigenvalues. So, for an $m-by-m$ matrix **H** where $m = 2n$, there exists n positive imaginary eigenvalues and n negative imaginary values. Therefore, in comparing exact to estimated eigenvalues, the comparison only requires n eigenvalues which is either all the positive imaginary eigenvalues or all the negative imaginary eigenvalues but not both. In all experiments, the positive imaginary eigenvalues were used.

Matlab's eig() function employs either LAPACK routines: SGEHRD or SHSEQR. It is not clear which LAPACK it will use.

To ensure high accuracy in the eig() function eigenvalue outputs, matrix **H** accuracy is extended through the Variable Precision Arithmetic (VPA) to 32 digits, vpa(H, 32). Applying vpa to matrix **H** increases the floating point decimal of the eigenvalues to an increased accurate level.

Procedure: Each numerical test examines a specific Matrix **H** structure. To determine The Method's eigenvalue behavior, the numerical test creates 100 randomly generated matrix **H** with the same structure. The base structure for developing matrix **H** is as follows:

1. 's' is a scalar parameter defining the submatrix **B**'s $\in \mathbf{R}^{2s\times 2s}$.
2. $\mathbf{B} = \tilde{\mathbf{B}} - \tilde{\mathbf{B}}^{\mathbf{T}}$ is a skew symmetric matrix. Using Matlab's uniform random matrix generator, rand(2s,2s), matrix $\tilde{\mathbf{B}}$ = rand(2s,2s). From the matrix generator, all entries, $\tilde{\mathbf{B}}_{\mathbf{ij}}$, have a uniform distribution between the numbers 0 and 1.
3. $\mathbf{C}_{\mathbf{2s\times 2s}}$ = rand(2s, 2s)+$(2s-1) * \mathbf{I}_{\mathbf{2s}}$; this algorithm ensures that submatrix **C** is a positive definite sub-matrix.[8]
4. $\mathbf{R}_{\mathbf{2s\times 2s}}$ = chol(C); Taking the Cholesky-Like factorization of submatrix **C** creates the upper triangular block in matrix **H**
5. $\mathbf{H}_{\mathbf{4s\times 4s}} = \begin{pmatrix} -\mathbf{B} & -\mathbf{R} \\ \mathbf{R}^{\mathbf{T}} & \mathbf{0} \end{pmatrix}$ This is the base structure for matrix **H** which is essentially characterized with having uniformly distributed entries in the sub-matrices **B**, **R**, and $\mathbf{R}^{\mathbf{T}}$.

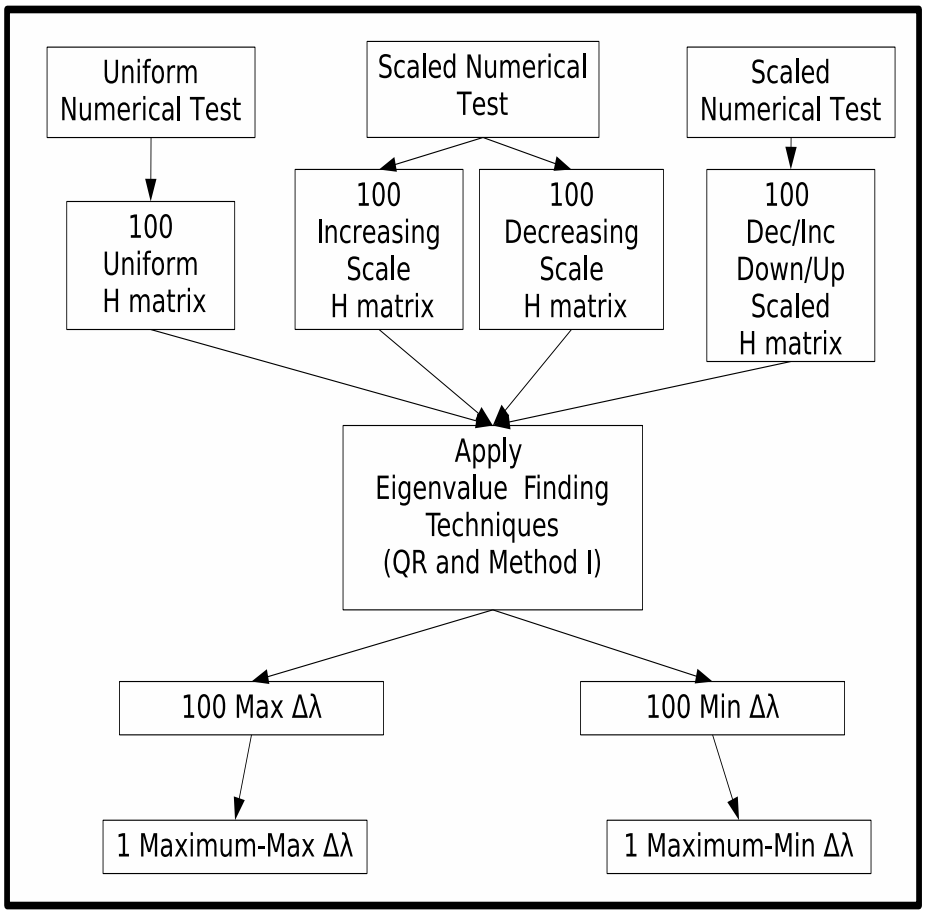
This uniform structured base matrix **H** is called the uniform **H** or uniform matrix **H**. This study then applies a variety of scaling factors to the uniform matrix **H** which preserves both the skew-symmetric properties as well as the inherent structure of $\mathbf{H} = \begin{bmatrix} -\mathbf{B} & -\mathbf{R} \\ \mathbf{R}^{\mathbf{T}} & \mathbf{0} \end{bmatrix}$. The scaling factors changes the scale of the entries of the submatrices, **B**, **R**, and $\mathbf{R}^{\mathbf{T}}$.

Therefore, testing the same matrix structure means that the same scaling

factors are applied to the uniform matrix $\mathbf{H}$'s submatrices $\mathbf{B}$, $\mathbf{R}$, and $\mathbf{R^T}$. In this study, seven matrix structures were numerically tested over a finite number of matrix dimensional sizes.

Data: Absolute error is calculated between the corresponding positive imaginary eigenvalues derived from eig()'s exact and The Method()'s estimated imaginary eigenvalues. For a given matrix $\mathbf{H}$ with dimension m, there are n positive imaginary eigenvalues. The absolute error for each positive imaginary eigenvalue is calculated and produces a total number of n absolute error values. For one matrix, the maximum absolute error and the minimum absolute error are then determined from the n absolute errors calculated from that specific matrix. After computations on 100 matrices with the same matrix structure, 100 maximum absolute (Max abs) errors and 100 minimum absolute (Min abs) errors are collected and stored as sample data. Figure 5.1 graphically depicts how these tests are run and data samples collected

Figure 5.1: Scheme of Numerical Test



From this sample data of 100 Max and Min abs errors the following datum are then calculated.

Max-Max: Extract the Maximum of the Maximum absolute error from the list of 100 Max abs errors.

Max-Min: Extract the Maximum of the Minimum absolute error from the list of 100 Min abs errors.

Limitations The matrix dimensions size are severely limited due to the computational load that comes with applying Matlab's eig() function to the Variable Precision Arithmetic (vpa) matrix **H**. This limit was deemed to be a matrix **H** of dimensions 60-by-60.

There are three numerical tests conducted in accordance to the experimental objective. They are the uniform matrix **H** test with no scale factor applied, the scale matrix **H** test which applies two scale factors, and the scaling submatrix **B** within matrix **H** test which applies four scaling factors. The description and results of these tests are as follows:

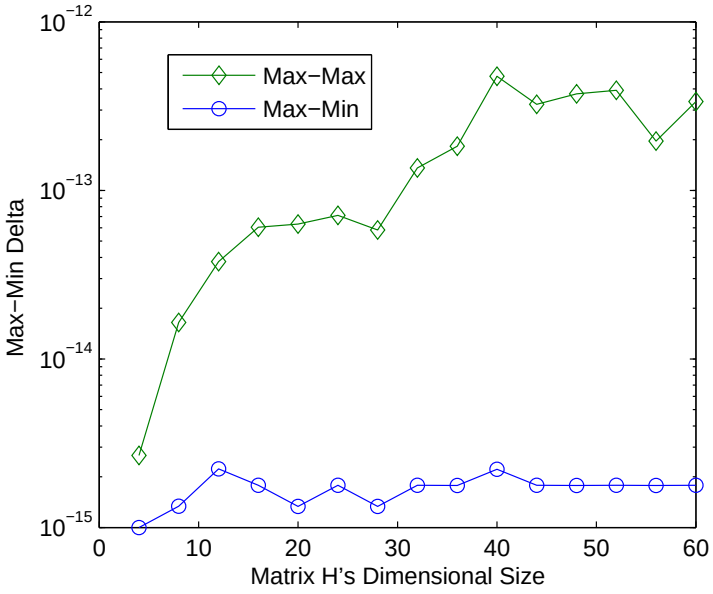
(1) Uniform matrix H Test: Using the uniform matrix **H** structure, this uniform test determines the data for a varying number of dimensional sizes of matrix **H**. No scaling factors applied. These dimensional sizes of **H** begin with 4-by-4 matrix, denoted as 4, and increased by 4 dimensions up to a dimensional size of 60. Each dimensional size of matrix **H** is then numerically tested and the data is extracted. The following table 5.1 depicts the data of the Uniform Test.

Table 5.1: Max-Min of a uniform matrix **H**

Matrix Dim	Max-Max $\Delta\lambda$	Max-Min $\Delta\lambda$
4	2.68 E-15	9.99 E-16
8	1.64 E-14	1.34 E-15
12	3.78 E-14	2.22 E-15
16	6.04 E-14	1.78 E-15
20	6.31 E-14	1.33 E-15
24	7.11 E-14	1.78 E-15
28	5.82 E-14	1.33 E-15
32	1.36 E-13	1.78 E-15
36	1.83 E-13	1.77 E-15
40	4.75 E-13	2.21 E-15
44	3.24 E-13	1.78 E-15
48	3.74 E-13	1.78 E-15
52	3.93 E-13	1.78 E-15
56	1.96 E-13	1.78 E-15
60	3.37 E-13	1.78 E-15

Based on data recorded in table 5.1, figure 5.2 graphically compares the Maximum-Max $\Delta\lambda$ and the Maximum-Min $\Delta\lambda$ error spread.

Figure 5.2: Numerical Uniform Test: Max-Max and Max-Min $\Delta\lambda$ Spread



In order to gauge whether these errors are theoretically correct, we compare this to a rudimentary upper bound limit. Since we consider the eigenvalues, λ, from eig(vpa($\mathbf{H}$)) as exact and The Method's eigenvalues, $\hat{\lambda}$'s as approximate, the $\Delta\lambda$ error analysis uses the absolute error and the relative error approach. Therefore, after the data for the uniform test is collected, a single uniform matrix $\mathbf{H}$ of dimensional size m is created and used to calculate a theoretical $\Delta\lambda$ error bound. A size m matrix $\mathbf{H}$ contains n positive pure imaginary eigenvalues from both eigenvalue procedures. From this approach, the theoretical absolute error bound for The Method is as follows:

$$|\hat{\lambda}_j - \lambda_j| \;\leq\; \texttt{C(m)*}\epsilon * ||\mathbf{H}||_{\mathbf{2}} \;\; \texttt{for} \;\; \mathbf{j} = \mathbf{1}, \mathbf{2}, \cdots, \mathbf{n}. \tag{5.1}$$

where Mach ϵ is machine precision equal to 1.1102×10^{-16} and C(m) is a certain low-degree polynomial of m. Since matrix $\mathbf{H}$ is a skew-symmetric

matrix, its 2-norm, $|\mathbf{H}|_{\mathbf{2}}$, is $\mathbf{H}$'s largest eigenvalue denoted as λ_1.

$$|\hat{\lambda}_j - \lambda_j| \ \leq \ \texttt{C(m)} * \epsilon * |\lambda_1| \ \ \forall \ \ j = 1, 2, \cdots, n. \tag{5.2}$$

From this, the relative error bound naturally follows:

$$\frac{|\hat{\lambda}_1 - \lambda_1|}{|\lambda_1|} \ \leq \ \texttt{C(m)*} \ \epsilon \ \ \forall \ \ \mathbf{j = 1}. \tag{5.3}$$

$$\frac{|\hat{\lambda}_2 - \lambda_2|}{|\lambda_2|} \ \leq \ \texttt{C(m)*} \ \epsilon * \frac{|\lambda_{\mathbf{1}}|}{|\lambda_{\mathbf{2}}|} \ \ \texttt{for} \ \ \mathbf{j = 2}.$$

$$\vdots$$

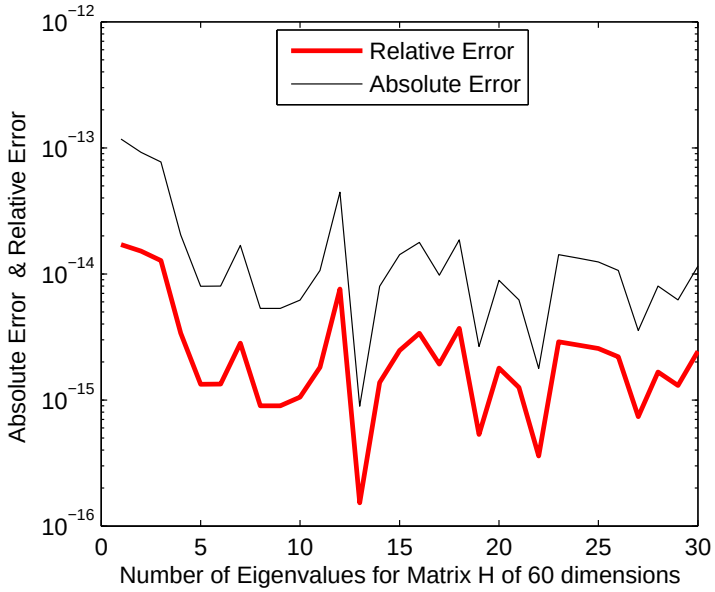
$$\frac{|\hat{\lambda}_n - \lambda_n|}{|\lambda_n|} \ \leq \ \texttt{C(m)*} \ \epsilon * \frac{|\lambda_{\mathbf{1}}|}{|\lambda_{\mathbf{n}}|} \ \ \texttt{for} \ \ \mathbf{j = n}. \tag{5.4}$$

where λ_n is matrix $\mathbf{H}$ smallest eigenvalue which implies that $|\frac{\lambda_1}{\lambda_n}|$ = Condition number of($\mathbf{H}$) = $\kappa(\mathbf{H})$. In our skew-symmetric $\mathbf{H}$ matrix, the maximum $|\lambda_1| = \sigma_1$ singular value and the minimum singular value $\sigma_n = |\lambda_n|$. In this test, a single uniform matrix $\mathbf{H}$ with $m = 60$ is used to determine λ_1 through λ_{30} and thus the theoretical error bound from λ_1 and λ_{30}. These two values considered exact from Matlab's eig() command are:

1. Maximum Eigenvalue $|\lambda_1| = 6.86475$
2. Minimum Eigenvalue $|\lambda_n| = |\lambda_{30}| = 4.74023$

Figure 5.3 then sketches out the relative error and absolute error curves for 30 eigenvalues extracted from this single uniform matrix $\mathbf{H}$ with $m = 60$. The first eigenvalue at point 1 is the λ_1 and the last point 30 corresponds to $|\lambda_{30}|$ where $|\lambda_1| \ \geq \ |\lambda_2| \ \geq \ \cdots \ \geq \ |\lambda_{30}|$

Figure 5.3: Relative λ error and absolute λ error from a single Uniform matrix $\mathbf{H}$ of 60 dimensions



$$10^{-12.9} \cong |\lambda_1 - \hat{\lambda}_1| \ \leq \ 165.2 * \ \epsilon * \ \lambda_{\mathbf{1}}.$$

From this single uniform matrix, C(m) = 165.2 is $\leq 3m$. So, the computed eigenvalues and their associated errors depicted in figures 5.2 and 5.3 satisfy the theoretical error bounds. With C($m = 60$) quite small, the error bound is then small which means we have a reasonable bound. However, this comparison is only from one single uniform matrix $\mathbf{H}$ so the small error bound is not statistically definitive. In addition, we took the largest matrix dimension of 60 and compared it to all the dimensions from 4 to 60. This also means that this test is not completely conclusive. Consequently, the test only indicates that the error bound appears to behave well for a uniform matrix $\mathbf{H}$.

(2) Scale matrix H Test: This numerical test scales the uniform matrix $\mathbf{H}$ using two different scaling congruence matrices. The scaling matrix either increases or decreases $\mathbf{B_{ij}}$ and $\mathbf{R_{ij}}$ and $\mathbf{R_{ij}^T}$ entries. For each scale, the test then measures the scaling effects on the $\Delta\lambda$ errors, determines whether the errors fall within a theoretical error bound or not, and indicates whether the theoretical error bound is reasonable.

The uniform matrix $\mathbf{H}$ is scaled in the following equation:

$$\text{Scaled } \widehat{\mathbf{H}} = \begin{pmatrix} \mathbf{D} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{pmatrix} \begin{pmatrix} -\mathbf{B} & -\mathbf{R} \\ \mathbf{R^T} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{D} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{pmatrix}$$

$$\Longrightarrow \text{ Scaled } \widehat{\mathbf{H}} = \begin{pmatrix} -\mathbf{D\,B\,D} & -\mathbf{D\,R} \\ \mathbf{R^T\,D} & \mathbf{0} \end{pmatrix},$$

where submatrix $\mathbf{I}$ is an identity matrix and submatrix $\mathbf{D}$ is a scaled identity matrix. Therefore, the scaled matrix $\mathbf{H}$ remains a skew-symmetric matrix with the sparse structure still preserved. Submatrix $\mathbf{D}$ is either an increasing or a decreasing scaling matrix. The building of these diagonal submatrices $\mathbf{D}$

is as follows.

$$\begin{aligned}
\mathbf{v} &= \{1, [1+1*7], [1+2*7], \ldots, [1+(p-1)*7], [1+p*7]\} \\
&\quad \texttt{and continues up to a maximum of p equaling to 14.} \\
\mathbf{v.}^{\mathbf{2}} &= \{v_1^2, v_2^2, v_3^2, \cdots, v_{k-1}^2, v_k^2\} \\
\mathbf{v.}^{\mathbf{2}} &= \{1, 64, 225, 484, 841, 1296, 1849, 2500, 3249, 4096, 5041, 6084, 7225, \\
&\quad 8464, 9801\} \\
\mathbf{g} &= \mathbf{v.}^{\mathbf{2}} \\
\mathbf{v.}^{\mathbf{-2}} &= \{v_1^{-2}, v_2^{-2}, v_3^{-2}, \cdots, v_{k-1}^{-2}, v_k^{-2}\} \\
\mathbf{v.}^{\mathbf{-2}} &= \{\ .016,\ .004,\ .002,\ .001,\ .0008,\ .0005, .0004,\ .0003,\ .0002,\ .000198, \\
&\quad .00016,\ .00014,\ .00012,\ .000102\} \\
\mathbf{g^{-1}} &= \mathbf{v.}^{-2}
\end{aligned}$$

So, for example, $\mathbf{g}(4) = 484$ and $\mathbf{g^{-1}}(4) = .001$ are scale factors at k = 4. Based on this, the scale diagonal matrix $\mathbf{D}$ is constructed.

$$\text{Increasing } \mathbf{D}(k) = \begin{pmatrix} \mathbf{g(k)} & 0 & \cdots & \cdots & 0 \\ 0 & \mathbf{g(k)} & 0 & \cdots & 0 \\ 0 & 0 & \ddots & \cdots & 0 \\ 0 & 0 & \cdots & \mathbf{g(k)} & 0 \\ 0 & 0 & \cdots & 0 & \mathbf{g(k)} \end{pmatrix}$$

$$\text{Decreasing } \mathbf{D}(k) = \begin{pmatrix} \mathbf{g^{-1}(k)} & 0 & \cdots & \cdots & 0 \\ 0 & \mathbf{g^{-1}(k)} & 0 & \cdots & 0 \\ 0 & 0 & \ddots & \cdots & 0 \\ 0 & 0 & \cdots & \mathbf{g^{-1}(k)} & 0 \\ 0 & 0 & \cdots & 0 & \mathbf{g^{-1}(k)} \end{pmatrix}$$

As an example of scaling matrix $\mathbf{D}$ of $\mathbf{H} \in \mathbf{R}^{8\times 8}$ with k=11,

where scaled $\widehat{\mathbf{H}} = \begin{pmatrix} -\mathbf{D^T}\ \mathbf{B}\ \mathbf{D} & -\mathbf{D^T}\ \mathbf{R} \\ \mathbf{R^T}\ \mathbf{D} & \mathbf{0} \end{pmatrix}$, the following scaling

matrix $\mathbf{D}$ would appear as follows:

1. Decreasing Scale

$$\mathbf{D}(11) = \begin{pmatrix} \mathbf{g^{-1}(11)} & 0 & 0 & 0 \\ 0 & \mathbf{g^{-1}(11)} & 0 & 0 \\ 0 & 0 & \mathbf{g^{-1}(11)} & 0 \\ 0 & 0 & 0 & \mathbf{g^{-1}(11)} \end{pmatrix}$$

$$= \begin{pmatrix} .000164 & 0 & 0 & 0 \\ 0 & .000164 & 0 & 0 \\ 0 & 0 & .000164 & 0 \\ 0 & 0 & 0 & .000164 \end{pmatrix}$$

2. Increasing Scale $\mathbf{D}(11) =$

$$\begin{pmatrix} \mathbf{g(11)} & 0 & 0 & 0 \\ 0 & \mathbf{g(11)} & 0 & 0 \\ 0 & 0 & \mathbf{g(11)} & 0 \\ 0 & 0 & 0 & \mathbf{g(11)} \end{pmatrix} = \begin{pmatrix} 5041 & 0 & 0 & 0 \\ 0 & 5041 & 0 & 0 \\ 0 & 0 & 5041 & 0 \\ 0 & 0 & 0 & 5041 \end{pmatrix}$$

For each increasing or decreasing scalar matrix $\mathbf{D}$, 100 random uniform matrix $\mathbf{H}$'s are created then scaled by the matrix $\mathbf{D}$. Once scaled, the Max-Max and Max-Min errors data are extracted. Due to the large amounts of data, this Scale Matrix $\mathbf{H}$ Test limits the matrix dimensions to 20, 40 and 60 dimensions.

Decreasing Scale: The decreasing Scale Matrix **H** Test depicts in tables 5.2 and 5.3 the sample data of the Max-max and Max-min $\Delta\lambda$ error spread.

Table 5.2: Max-Min of a decreasing scale matrix **H** with dimensions 20 and 40

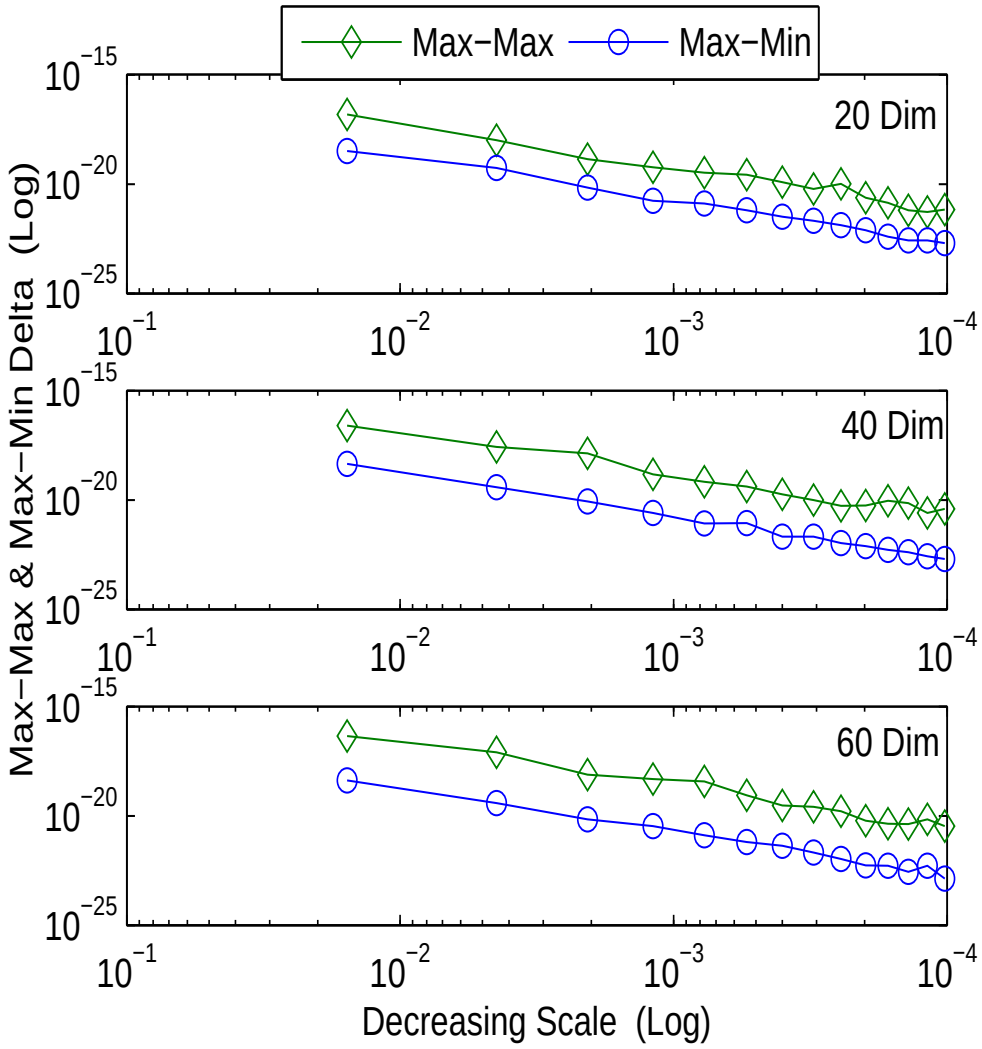
	Matrix 20x20		Matrix 40x40	
Scale Size	Max-Max $\Delta\lambda$	Max-Min $\Delta\lambda$	Max-Max $\Delta\lambda$	Max-Min $\Delta\lambda$
0.015625	1.52E-17	3.26E-19	2.52E-17	4.52E-19
0.004444	9.96E-19	5.42E-20	2.68E-18	3.85E-20
0.002066	1.39E-19	6.76E-21	1.33E-18	8.47E-21
0.001189	5.76E-20	1.71E-21	1.49E-19	2.53E-21
0.000772	3.3E-20	1.27E-21	6.65E-20	8.53E-22
0.000541	2.65E-20	6.36E-22	4.09E-20	8.61E-22
0.0004	1.22E-20	3.17E-22	1.81E-20	2.13E-22
0.000308	5.98E-21	2.13E-22	9.85E-21	2.11E-22
0.000244	1.01E-20	1.33E-22	5.35E-21	1.05E-22
0.000198	2.38E-21	7.83E-23	5.56E-21	7.85E-23
0.000164	1.38E-21	3.96E-23	9.11E-21	5.3E-23
0.000138	6.22E-22	2.66E-23	7.24E-21	4.08E-23
0.000118	5.29E-22	2.65E-23	2.51E-21	2.65E-23
0.000102	6.82E-22	1.98E-23	3.92E-21	1.98E-23

Table 5.3: Max-Min of a decreasing scale matrix **H** with dimension 60

	Matrix 60x60	
Scale Size	Max-Max $\Delta\lambda$	Max-Min $\Delta\lambda$
0.015625	4.4E-17	4.19E-19
0.004444	7.91E-18	3.86E-20
0.002066	7.52E-19	6.85E-21
0.001189	4.83E-19	3.39E-21
0.000772	3.67E-19	1.27E-21
0.000541	8.51E-20	6.36E-22
0.0004	3E-20	4.23E-22
0.000308	2.58E-20	2.1E-22
0.000244	1.61E-20	1.06E-22
0.000198	5.93E-21	5.43E-23
0.000164	4.34E-21	5.27E-23
0.000138	4.1E-21	2.7E-23
0.000118	6.94E-21	5.29E-23
0.000102	3.36E-21	1.33E-23

Based on data recorded in tables 5.2 and 5.3, figure 5.4 graphically compares the Maximum-Max $\Delta\lambda$ and the Maximum-Min $\Delta\lambda$ error spread.

Figure 5.4: Decreasing Scale of matrix **H** Test: Max-Min $\Delta\lambda$ Error Spread of a 20, 40 and 60 matrix **H** sizes



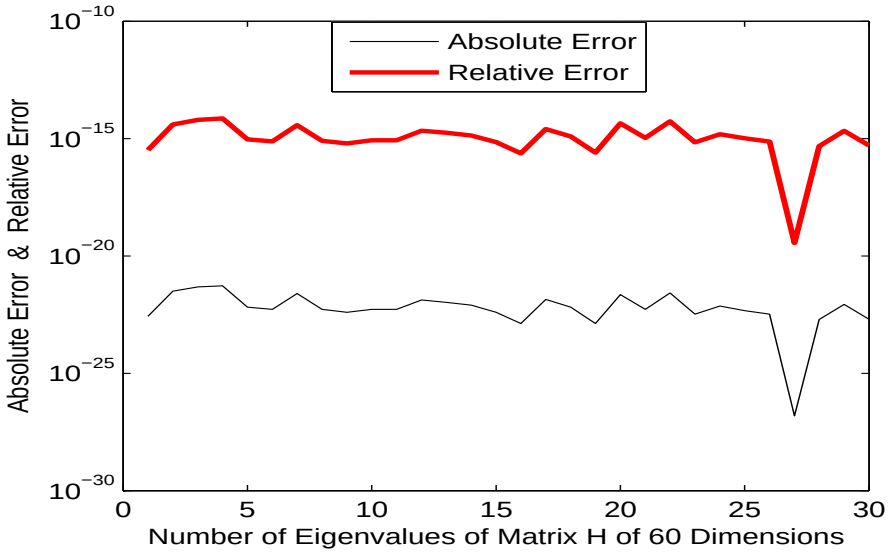
For error bound analysis, a decreasing scale matrix is applied to a single uniform matrix **H** of dimension 60 and the theoretical error bound is determined. Although it would be preferable to determine the error bounds for all fifteen

scale matrices, this study selected the smallest decreasing scale matrix for the error bound analysis. Submatrix $\mathbf{D}(14)$ takes the form shown below.

$$\mathbf{D}(14)= \begin{pmatrix} .000102 & 0 & 0 & 0 & 0 \\ 0 & .000102 & 0 & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & .000102 & 0 \\ 0 & 0 & 0 & 0 & .000102 \end{pmatrix}_{30\times 30} .$$

Figure 5.5 sketches out the relative error and absolute error curves for 30 eigenvalues extracted from a decreasing scaled matrix $\mathbf{H} = \widehat{\mathbf{H}}$ of size 60. The first eigenvalue at point 1 is the λ_1 and the last point 30 corresponds to $|\lambda_{30}|$ where $|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_{30}|$

Figure 5.5: Relative λ error and absolute λ error from a single decreasing scaled matrix $\mathbf{H}$ of 60 dimensions



From the single matrix $\widehat{\mathbf{H}}$ of size 60, Matlab's eig() extracts the exact max

and min eigenvalues λ_1 and λ_{30} respectfully, for determining the theoretical error bound. These two eigenvalues are:

1. Maximum Eigenvalue $|\lambda_1| = .8241 \times 10^{-7}$
2. Minimum Eigenvalue $|\lambda_n| = |\lambda_{30}| = .3871 \times 10^{-7}$

From these equivalent singular values of $\widehat{\mathbf{H}}$, the theoretical eigenvalue bounds, eqn (5.1), are:

$$10^{-16} \cong |\hat{\lambda}_1 - \lambda_1| \ \leq \ (1.0929 \times 10^7) * \ \epsilon * \ |\lambda_{\mathbf{1}}|.$$

In this case, C($m = 60$) is roughly 5 orders of magnitude of m, C($m = 60$) = 1.0929×10^7 which is $< 1.83 * 10^5 m$. C(m) is quite large and is not a reasonable bound. Without further tests, one possible answer for such a large error bound is due to severe cancellation errors. The entries were severely scaled down and these small entries could have created some severe cancellations during the application of the algorithms.

Clearly, from figure 5.4 the computed eigenvalues and their associated errors satisfy the theoretical error bound. Again, this conclusion is not definitive. Using only one scaled matrix **H** is not statistically sufficient. In addition, only one scale was used instead of all 15 scales.

Increasing Scale: The increasing Scale Matrix **H** Test depicts in tables 5.4 and 5.5 the sample data of the Max-Max and Max-Min $\Delta\Lambda$ error spread.

Table 5.4: Max-Min of an increasing scale matrix **H** with dimensions 20 and 40

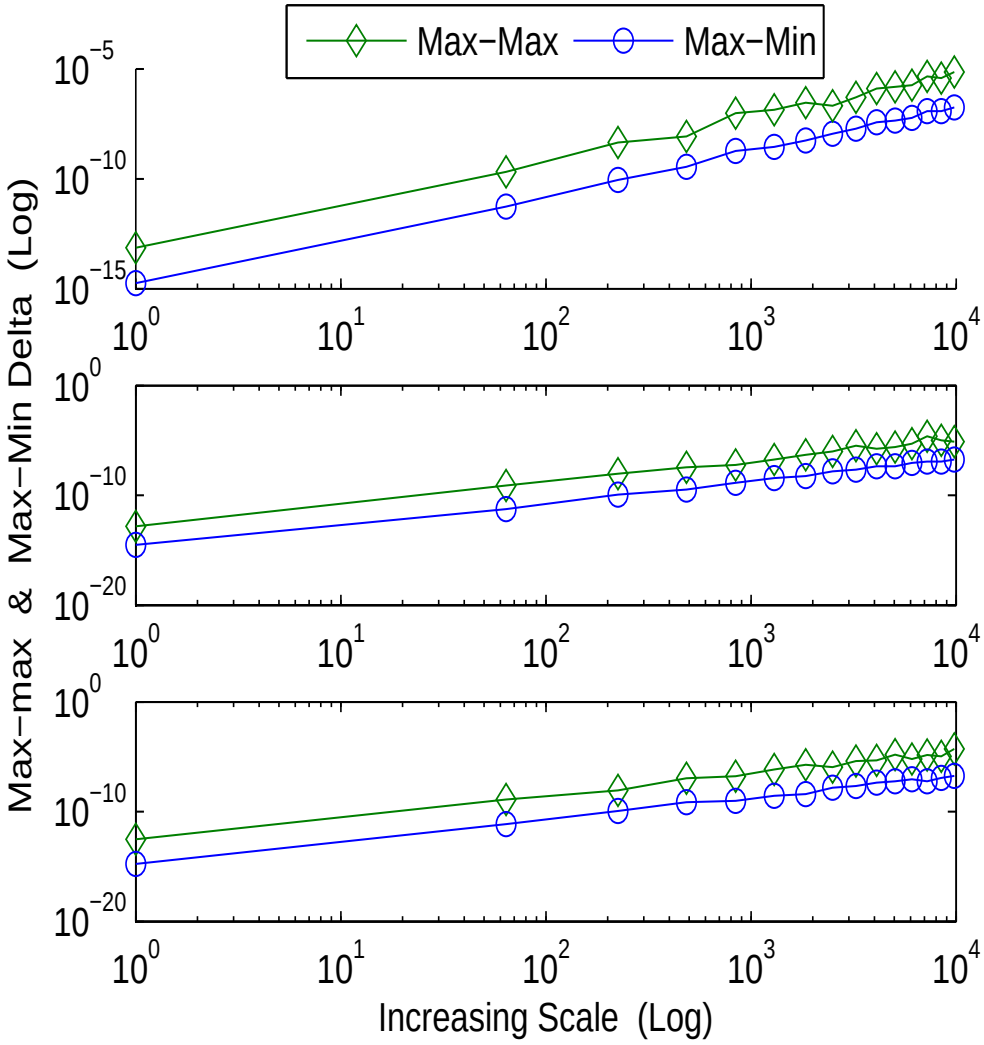
	Matrix 20x20		Matrix 40x40	
Scale Size	Max-Max $\Delta\lambda$	Max-Min $\Delta\lambda$	Max-Max $\Delta\lambda$	Max-Min $\Delta\lambda$
1	7.28E-14	1.77E-15	1.47E-13	3.11E-15
64	2.07E-10	5.49E-12	7.86E-10	5.44E-12
225	4.5E-09	8.81E-11	9.2E-09	1.17E-10
484	8.38E-09	3.49E-10	3.54E-08	3.44E-10
841	9.64E-08	1.84E-09	6.01E-08	1.39E-09
1296	1.38E-07	2.8E-09	1.81E-07	3.73E-09
1849	2.85E-07	5.58E-09	4.81E-07	5.57E-09
2500	2.09E-07	1.12E-08	1.02E-06	1.49E-08
3249	4.99E-07	1.86E-08	3.26E-06	2.24E-08
4096	1.27E-06	3.73E-08	1.68E-06	4.46E-08
5041	1.52E-06	4.47E-08	2.41E-06	4.5E-08
6084	1.77E-06	5.94E-08	5.04E-06	8.87E-08
7225	4.47E-06	1.19E-07	2.36E-05	1.18E-07
8464	3.75E-06	1.19E-07	9.95E-06	1.19E-07
9801	7.21E-06	1.78E-07	7.45E-06	1.79E-07

Table 5.5: Max-Min of an increasing scale matrix **H** with dimension 60

	Matrix 60x60	
Scale Size	Max-Max $\Delta\lambda$	Max-Min $\Delta\lambda$
1	3.09E-13	1.78E-15
64	1.27E-09	7.31E-12
225	8.67E-09	1.16E-10
484	1.12E-07	7.01E-10
841	1.68E-07	9.4E-10
1296	6.91E-07	2.78E-09
1849	1.84E-06	3.83E-09
2500	1.12E-06	1.49E-08
3249	3.93E-06	2.24E-08
4096	4.69E-06	4.46E-08
5041	1.54E-05	5.87E-08
6084	6.26E-06	8.9E-08
7225	1.46E-05	6.01E-08
8464	1.07E-05	1.2E-07
9801	5.19E-05	1.79E-07

Based on the sample data in tables 5.4 and 5.5, figure 5.6 graphically compares the Maximum-Max $\Delta\lambda$ and the Maximum-Min $\Delta\lambda$ error spread.

Figure 5.6: Increasing Scale of matrix **H** Test: Max-Min $\Delta\lambda$ error Spread of a 20, 40 and 60 matrix **H** sizes



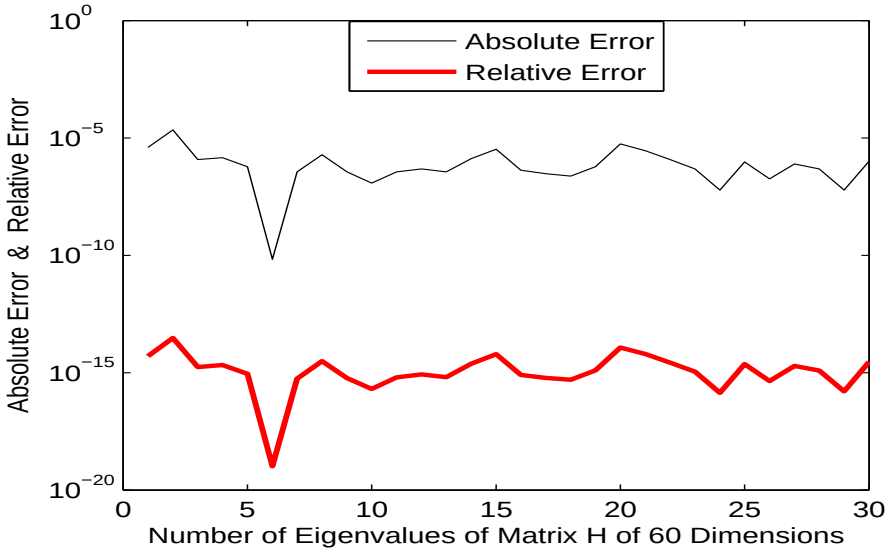
For error bound analysis, an increasing scale matrix is applied to a single

uniform Matrix $\mathbf{H}$ of dimensions 60. From this increased scaled matrix $\mathbf{H}$, the theoretical error bound is then determined. Again, it would be preferable to determine the error bounds for all fifteen scale matrices. However, due to limited time, this study selected the largest increasing scale matrix for the error bound analysis. Submatrix $\mathbf{D}(15)$ takes the form shown below:

$$\mathbf{D}(15)= \begin{pmatrix} 9801 & 0 & 0 & 0 & 0 \\ 0 & 9801 & 0 & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & 9801 & 0 \\ 0 & 0 & 0 & 0 & 9801 \end{pmatrix}_{30\times 30},$$

Figure 5.7 sketches out the relative error and absolute error curves for 30 eigenvalues extracted from a increasing scaled matrix $\mathbf{H} = \widehat{\mathbf{H}}$'s of size 60. The first eigenvalue at point 1 is the λ_1 and the last point 30 corresponds to $|\lambda_{30}|$ where $|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_{30}|$.

Figure 5.7: Relative λ Error and absolute λ error from a single increasing scaled matrix of 60 dimensions



From this single matrix $\widehat{\mathbf{H}}$, Matlab's function eig() extracts the exact max and min eigenvalues, λ_1 and λ_{30} respectfully, for determining the theoretical error bound. These two eigenvalues are:

1. Maxiumum Eigenvalue $|\lambda_1| = 777137439 \approx 7.77 \times 10^8$
2. Minimum Eigenvalue $|\lambda_n| = |\lambda_{30}| = 351932045 \approx 3.52 \times 10^8$

From these equivalient singular values of $\widehat{\mathbf{H}}$, the theoretical Eigenvalue bounds, eqn (5.1), are:

$$10^{-6} \cong |\hat{\lambda}_1 - \lambda_1| \ \leq \ (11.6) * \ \epsilon \ \lambda_{\mathbf{1}}.$$

In this case, C($m = 60$) is less than one order of magnitude m, C($m = 60$) = 11.6, which is $< .2m$. C(m) is less than a multiple of m which is considered

quite small and therefore a reasonable error bound. From figure 5.6, the sample Max-Min absolute error does satisfy the theoretical error bound. However, the sample Max-Max absolute error data does not satisfy it. In order to bound both absolute error sample data, C(m) would have to be increased in size by 15 or 1.5 order of magnitude. So, C(m) $< 3m$ is required to bound the data which is still quite small and therefore reasonable. Unfortunately, this error bound is not definitive. Using only one scaled matrix **H** is not statistically sufficient to make a definitive conclusion. In addition, only one scale was used instead of determining all the theoretical error bounds from 15 scales over a large sample of uniform matrix **H**s.

(3) Scaling submatrix B Matrix within matrix H Test

This numerical test scales only the submatrix **B** within the uniform matrix **H**. There are four different scaling processes: increases, decreases, up or down. The scaling matrix only scales the submatrix **B**. Since the scaling matrix is diagonal, it is therefore a special congruence transformation matrix which preserves both the skew-symmetric and sparse structure of matrix **H**.

The scaling of submatrix **B** within the uniform matrix **H** Test (Scale **B** in **H** Test) is constructed in the following equation:

$$\widehat{\mathbf{B}} = \mathbf{D}\ \mathbf{B}\ \mathbf{D} \Longrightarrow \widehat{\mathbf{H}} = \begin{pmatrix} -\widehat{\mathbf{B}} & -\mathbf{R} \\ \mathbf{R^T} & \mathbf{0} \end{pmatrix} \tag{5.5}$$

Therefore, the scaled matrix **H** remains a skew-symmetric matrix with the sparse structure still preserved. Submatrix **D** is either an decreasing, increasing, down, or up scaling matrix. The building of the decreasing and increasing diagonal submatrices **D** is the same as the previous Scale Matrix **H** Test. How-

ever, instead of the scaling matrix structured as $\begin{pmatrix} \mathbf{D_{n\times n}} & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{pmatrix}$ it is simply $\mathbf{D_{n\times n}}$. Again, just like the Scale Matrix $\mathbf{H}$ Test, the building of increasing or decreasing diagonal submatrices $\mathbf{D}$ is shown below:

$$\begin{aligned}
\mathbf{v} &= \{1, [1+1*7], [1+2*7], \ldots, [1+(p-1)*7], [1+p*7]\} \\
&\quad \texttt{and continues up to a maximum of p equaling to 14.} \\
\mathbf{v.}^{\mathbf{2}} &= \{v_1^2, v_2^2, v_3^2, \cdots, v_{k-1}^2, v_k^2\} \\
\mathbf{v.}^{\mathbf{2}} &= \{1, 64, 225, 484, 841, 1296, 1849, 2500, 3249, 4096, 5041, 6084, 7225, \\
&\quad 8464, 9801\} \\
\mathbf{g} &= \mathbf{v.}^{\mathbf{2}} \\
\mathbf{v.}^{\mathbf{-2}} &= \{v_1^{-2}, v_2^{-2}, v_3^{-2}, \cdots, v_{k-1}^{-2}, v_k^{-2}\} \\
\mathbf{v.}^{\mathbf{-2}} &= \{\ .016,\ .004,\ .002,\ .001,\ .0008,\ .0005, .0004,\ .0003,\ .0002,\ .000198, \\
&\quad .00016,\ .00014,\ .00012,\ .000102\} \\
\mathbf{g^{-1}} &= \mathbf{v.}^{-2}
\end{aligned}$$

$$\text{decreasing } \mathbf{D}(k) = \begin{pmatrix} \mathbf{g^{-1}}(k) & 0 & \cdots & \cdots & 0 \\ 0 & \mathbf{g^{-1}}(k) & 0 & \cdots & 0 \\ 0 & 0 & \ddots & \cdots & 0 \\ 0 & 0 & \cdots & \mathbf{g^{-1}}(k) & 0 \\ 0 & 0 & \cdots & 0 & \mathbf{g^{-1}}(k) \end{pmatrix}$$

$$\text{Increasing } \mathbf{D}(k) = \begin{pmatrix} \mathbf{g}(k) & 0 & \cdots & \cdots & 0 \\ 0 & \mathbf{g}(k) & 0 & \cdots & 0 \\ 0 & 0 & \ddots & \cdots & 0 \\ 0 & 0 & \cdots & \mathbf{g}(k) & 0 \\ 0 & 0 & \cdots & 0 & \mathbf{g}(k) \end{pmatrix}$$

Using the entries $\mathbf{g}(\mathrm{k})$ or $\mathbf{g^{-1}}(\mathrm{k})$ $\forall$ $\mathbf{k}$ $\in$ $\{1, 2, 3, ..., 15\}$ to create a diagonal submatrix $\mathbf{D}$, submatrix $\mathbf{B}$ is then scaled 15 times. Essentially, the increasing and decreasing matrix $\mathbf{D}$ uniformly increases or decreases the b_{ij} elements by $\mathbf{g^2}(\mathrm{k})$ and $\mathbf{g^{-1^2}}(\mathrm{k})$ respectively. Again, this is essentially the same concept as the scaling matrix used in the Scale Matrix $\mathbf{H}$ Test.

The up and down scaling matrices $\mathbf{D}$ are slightly different. The diagonal entries for $\mathbf{D}$ are not uniformly scaled. For a matrix $\mathbf{B}$ of size n, The building of a down scale $\mathbf{D}$ is as follows:

$$\begin{aligned} \mathbf{g_k^{-1}} &= \{1^{-}k,\ 2^{-}k,\ 3^{-}k,\ ...\ 14^{-}k\} \texttt{ for k =1} \\ \mathbf{g_k^{-1}} &= \{1^{-k/6},\ 2^{-k/6},\ 3^{-k/6},\ ...\ 14^{-k/6}\} \texttt{ for k = 2, 3, ... 14.} \end{aligned}$$

$$\text{down scale matrix } \mathbf{D}(k) = \begin{pmatrix} \mathbf{g}_k^{-1}(1) & 0 & \cdots & \cdots & 0 \\ 0 & \mathbf{g}_k^{-1}(2) & 0 & \cdots & 0 \\ 0 & 0 & \ddots & \cdots & 0 \\ 0 & 0 & \cdots & \mathbf{g}_k^{-1}(n-1) & 0 \\ 0 & 0 & \cdots & 0 & \mathbf{g}_k^{-1}(n) \end{pmatrix}$$

Concurrently, the building of a up scale $\mathbf{D}$ is as follows:

$$\begin{aligned} \mathbf{g_k^{1}} &= \{1^{k},\ 2^{k},\ 3^{k},\ ...\ 14^{k}\} \texttt{ for k =1} \\ \mathbf{g_k^{1}} &= \{1^{k/6},\ 2^{k/6},\ 3^{k/6},\ ...\ 14^{k/6}\} \texttt{ for k = 2, 3, ... 14.} \end{aligned}$$

$$\text{up scale matrix } \mathbf{D}(k) = \begin{pmatrix} \mathbf{g}_k^1(1) & 0 & \cdots & \cdots & 0 \\ 0 & \mathbf{g}_k^1(2) & 0 & \cdots & 0 \\ 0 & 0 & \ddots & \cdots & 0 \\ 0 & 0 & \cdots & \mathbf{g}_k^1(n-1) & 0 \\ 0 & 0 & \cdots & 0 & \mathbf{g}_k^1(n) \end{pmatrix}$$

As an example of the scaling matrix $\mathbf{D}$ of the submatrix $\mathbf{B} \in \mathbf{R}^{4\times 4}$ with k=8, where $\widehat{\mathbf{B}} = \mathbf{D}\ \mathbf{B}\ \mathbf{D}$, the following scaling matrix $\mathbf{D}$ would appear as follows:

1. Decreasing Scale $\mathbf{D}(8) =$

$$\begin{pmatrix} G^{-1}(8) & 0 & 0 & 0 \\ 0 & G^{-1}(8) & 0 & 0 \\ 0 & 0 & G^{-1}(8) & 0 \\ 0 & 0 & 0 & G^{-1}(8) \end{pmatrix} = \begin{pmatrix} .00031 & 0 & 0 & 0 \\ 0 & .00031 & 0 & 0 \\ 0 & 0 & .00031 & 0 \\ 0 & 0 & 0 & .00031 \end{pmatrix}$$

2. Increasing Scale $\mathbf{D}(8) =$

$$\begin{pmatrix} G(8) & 0 & 0 & 0 \\ 0 & G(8) & 0 & 0 \\ 0 & 0 & G(8) & 0 \\ 0 & 0 & 0 & G(8) \end{pmatrix} = \begin{pmatrix} 2500 & 0 & 0 & 0 \\ 0 & 2500 & 0 & 0 \\ 0 & 0 & 2500 & 0 \\ 0 & 0 & 0 & 2500 \end{pmatrix}$$

3. Down Scale $\mathbf{D}(8) =$

$$\begin{pmatrix} G_8(1) & 0 & 0 & 0 \\ 0 & G_8(2) & 0 & 0 \\ 0 & 0 & G_8(3) & 0 \\ 0 & 0 & 0 & G_8(4) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & .397 & 0 & 0 \\ 0 & 0 & .231 & 0 \\ 0 & 0 & 0 & .158 \end{pmatrix}$$

4. Up Scale $\mathbf{D}(8) =$

$$\begin{pmatrix} G_8(1) & 0 & 0 & 0 \\ 0 & G_8(2) & 0 & 0 \\ 0 & 0 & G_8(3) & 0 \\ 0 & 0 & 0 & G_8(4) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2.52 & 0 & 0 \\ 0 & 0 & 4.33 & 0 \\ 0 & 0 & 0 & 6.35 \end{pmatrix}$$

For each "k" index parameter which defines the decreasing, increasing, down, and up scalar submatrix $\mathbf{D}(k)$, 100 random uniform matrix $\mathbf{H}$'s are created and their associated submatrix $\mathbf{B}$ are scaled by $\mathbf{D}(k)$. Once scaled, the sample Max-max and Max-min error data are extracted from matrix $\mathbf{H}$. Again, due to the large amounts of data, Scale $\mathbf{B}$ in $\mathbf{H}$ Test limits the matrix dimensions to 20, 40 and 60 dimensions.

Decreasing Scale B: In the decreasing Scale $\mathbf{B}$ in $\mathbf{H}$ Test depicts in tables 5.6 to 5.7 the sample data of Max-Max and Max-Min error spread.

Table 5.6: Max-Min of a decreasing scale submatrix $\widehat{\mathbf{B}}$ of matrix $\mathbf{H}$ with dim 20 and 40

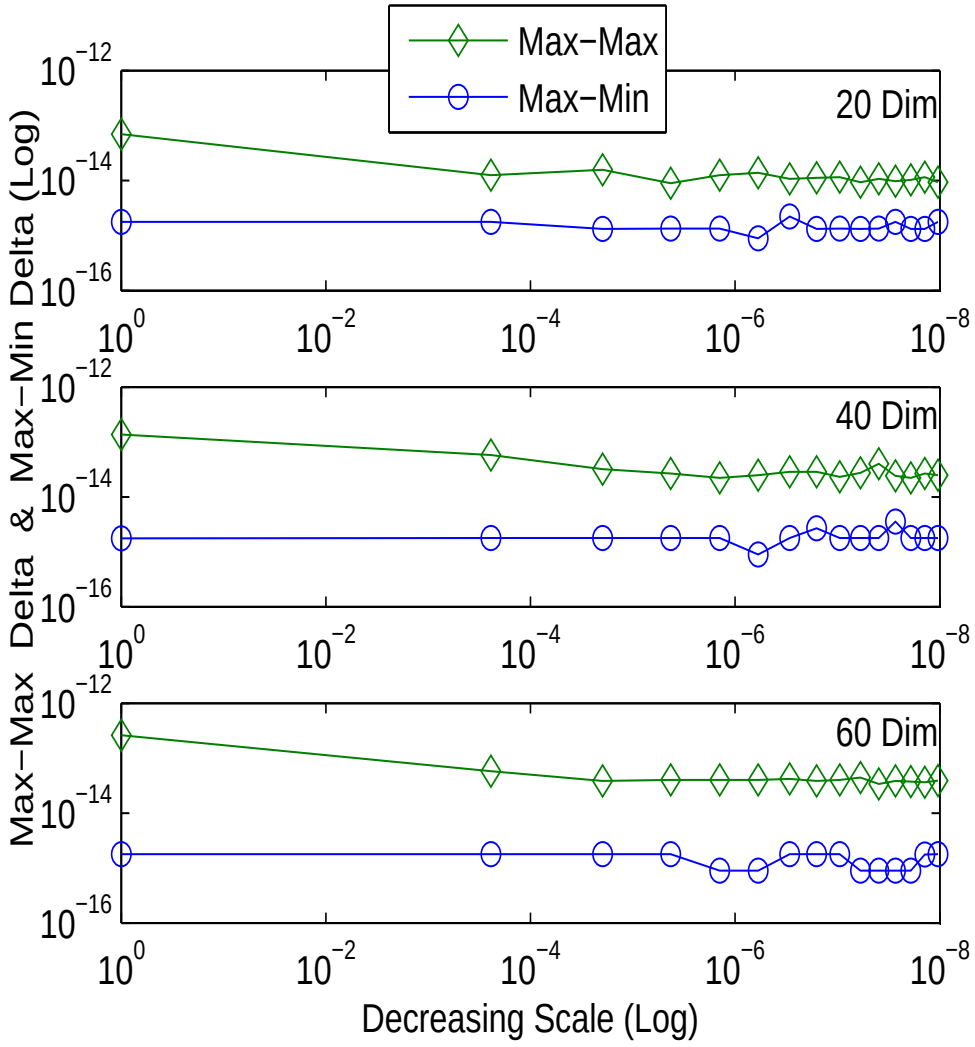
	Matrix 20x20		Matrix 40x40	
Scale	Max-Max $\Delta\lambda$	Max-Min $\Delta\lambda$	Max-Max $\Delta\lambda$	Max-Min $\Delta\lambda$
0.000244	1.24E-14	1.77E-15	5.77E-14	1.78E-15
1.98E-05	1.58E-14	1.33E-15	3.20E-14	1.77E-15
4.27E-06	8.89E-15	1.34E-15	2.66E-14	1.77E-15
1.41E-06	1.25E-14	1.34E-15	2.22E-14	1.78E-15
5.95E-07	1.38E-14	8.91E-16	2.49E-14	8.99E-16
2.93E-07	1.07E-14	2.21E-15	2.84E-14	1.78E-15
1.6E-07	1.11E-14	1.33E-15	2.84E-14	2.66E-15
9.47E-08	1.16E-14	1.34E-15	2.31E-14	1.78E-15
5.96E-08	9.33E-15	1.32E-15	2.75E-14	1.78E-15
3.94E-08	1.07E-14	1.33E-15	4.00E-14	1.78E-15
2.7E-08	9.76E-15	1.77E-15	2.40E-14	3.55E-15
1.92E-08	1.02E-14	1.33E-15	2.22E-14	1.77E-15
1.4E-08	1.15E-14	1.32E-15	2.67E-14	1.77E-15
1.04E-08	9.34E-15	1.77E-15	2.49E-14	1.77E-15

Table 5.7: Max-Min of a decreasing scale submatrix $\widehat{\mathbf{B}}$ of matrix $\mathbf{H}$ with dim 60

	Matrix 60x60	
Scale	Max-Max $\Delta\lambda$	Max-Min $\Delta\lambda$
0.000244	5.77E-14	1.77E-15
1.98E-05	3.82E-14	1.77E-15
4.27E-06	4E-14	1.78E-15
1.41E-06	4E-14	8.93E-16
5.95E-07	4E-14	8.94E-16
2.93E-07	4.17E-14	1.78E-15
1.6E-07	3.82E-14	1.78E-15
9.47E-08	4E-14	1.77E-15
5.96E-08	4.44E-14	8.95E-16
3.94E-08	3.38E-14	8.96E-16
2.7E-08	3.82E-14	8.93E-16
1.92E-08	3.73E-14	8.92E-16
1.4E-08	3.64E-14	1.76E-15
1.04E-08	3.91E-14	1.78E-15

Based on the sample data recorded in tables 5.6 to 5.7, figure 5.8 graphically compares the Maximum-Max $\Delta\lambda$ and the Maximum-Min $\Delta\lambda$ error spread.

Figure 5.8: Decreasing Scale of submatrix **B** Test: Max-Min $\Delta\lambda$ Error Spread of a 20, 40 and 60 matrix **H** sizes



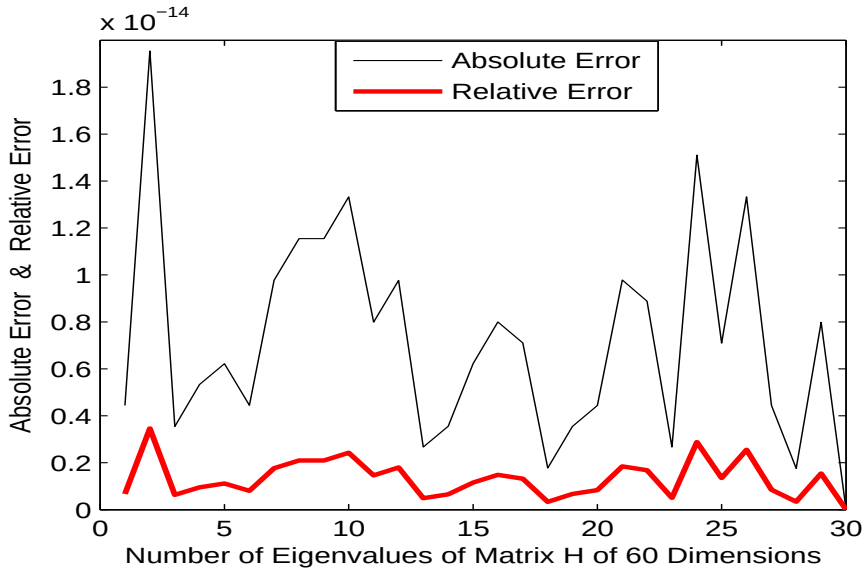
For error bound analysis, a decreasing scale matrix **D** is applied to submatrix

B in a single matrix **H** with dimension 60. From this single scaled matrix **H**, the theoretical error bound is determined. As in the other previous scaled tests, it would be preferable to apply all fifteen scales. However, time constraints confines this study to analyze the impacts of the smallest scale matrix. Submatrix $\mathbf{D}(14)$ takes the form shown below:

$$\mathbf{D}(14)= \begin{pmatrix} .000102 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & .000102 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & .000102 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \ddots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & .000102 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & .000102 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & .000102 \end{pmatrix}_{30\times 30}.$$

Figure 5.9 sketches out the relative error and absolute error curves for 30 eigenvalues extracted from a decreasing scaled submatrix $\widehat{\mathbf{B}}$ within a matrix $\widehat{\mathbf{H}}$ of size 60. The first eigenvalue at point 1 is $|\lambda_1|$ and the last point 30 corresponds to $|\lambda_{30}|$ where $|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_{30}|$.

Figure 5.9: Relative λ error and absolute λ error from a scaled submatrix $\mathbf{B}$ within a single matrix $\mathbf{H}$ of dimension 60



From this single matrix $\widehat{\mathbf{H}}$ of size 60, Matlab's extracts the exact max and min eigenvalues, λ_1 and λ_{30} respectfully, for determining the theoretical error bound. These two eigenvalues are:

1. Maximum Eigenvalue $|\lambda_1| \cong 6.62$
2. Minimum Eigenvalue $|\lambda_n| = |\lambda_{30}| \cong 5.17$

From these equivalent singular values of $\widehat{\mathbf{H}}$, the theoretical eigenvalue bounds, eqn (5.1), are:

$$.4442 \times 10^{-14} \cong |\hat{\lambda}_1 - \lambda_1| \ \leq \ 6.04 * \ \epsilon * \ |\lambda_{\mathbf{1}}|.$$

In this case, C($m = 60$) is less than one order of magnitude of m, C($m = 60$) = 6.04, which is $< .1m$. C(m) is quite small which makes it a reasonable error bound.

From figure 5.8, the sample Max-Min absolute error does satisfy the theoretical error bound. However, the sample Max-Max absolute error data does not satisfy the error bound. In order to fully bound the sample absolute error data, C(m) must be increased by an order of two. This would mean that C(m)$< 10m$ which is still small and therefore a reasonable bound. Since this is a smallest decreasing scale applied, just like in the Scale Matrix **H** Test, the scaled elements b_{ij} might have severe cancel effects. Again, this theoretical error bound is neither definitive nor comprehensive. The study used only one matrix **H** in calculating the theoretical bound, so it is not statistically sufficient to make definitive conclusions. In addition, only one scale matrix was used out of the fifteen available which makes this study not completely comprehensive. Essentially, this study indicates that the error bound appears reasonable under this most severe scaling condition.

#2 Increasing Scale B: The increasing Scale **B** in **H** Test depicts in tables 5.8 to 5.9 the sample data of the Max-Max and Max-Min error spread.

Table 5.8: Max-Min of an increasing scale **B** of matrix **H** with dim 20 and 40

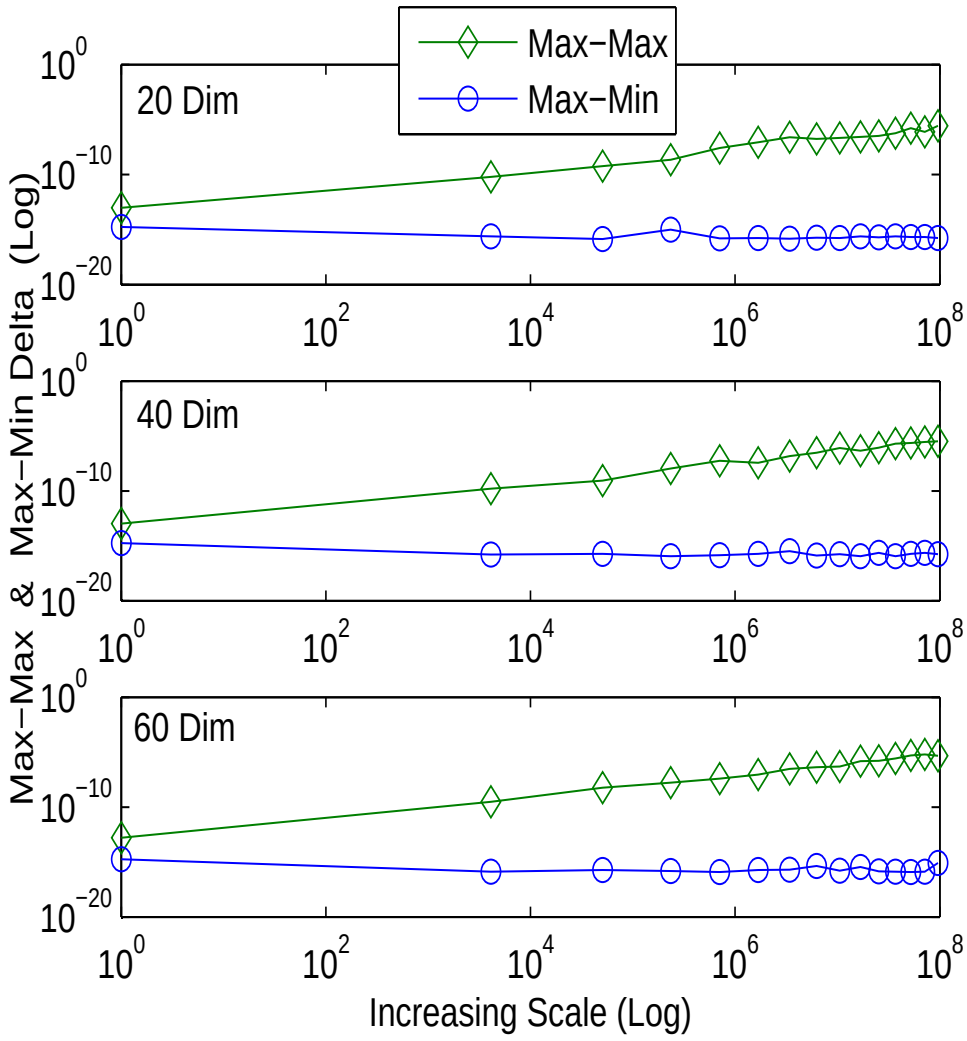
	Matrix 20x20		Matrix 40x40	
Scale	Max-Max $\Delta\lambda$	Max-Min $\Delta\lambda$	Max-Max $\Delta\lambda$	Max-Min $\Delta\lambda$
1	9.77E-14	1.78E-15	1.03E-13	1.77E-15
4096	6E-11	2.36E-16	1.63E-10	1.55E-16
50625	5.89E-10	1.38E-16	8.74E-10	1.82E-16
234256	2.24E-09	9.68E-16	1.1E-08	1.16E-16
707281	2.65E-08	1.53E-16	5.43E-08	1.44E-16
1679616	8.29E-08	1.7E-16	3.63E-08	1.8E-16
3418801	2.51E-07	1.42E-16	1.47E-07	3.21E-16
6250000	1.68E-07	1.87E-16	2.87E-07	1.27E-16
10556001	2.12E-07	1.71E-16	7.97E-07	1.68E-16
16777216	2.65E-07	2.4E-16	4.36E-07	1.12E-16
25411681	3.2E-07	1.98E-16	8.27E-07	2.3E-16
37015056	5.81E-07	2.44E-16	1.92E-06	1.09E-16
52200625	1.59E-06	2.06E-16	2.35E-06	1.8E-16
71639296	7.76E-07	2.03E-16	2.86E-06	2.26E-16
96059601	2.65E-06	1.73E-16	3.34E-06	1.75E-16

Table 5.9: Max-Min of an increasing scale **B** of matrix **H** with dim 60

	Matrix 60x60	
Scale	Max-Max $\Delta\lambda$	Max-Min $\Delta\lambda$
1	1.61E-13	1.78E-15
4096	2.91E-10	1.32E-16
50625	5.91E-09	1.87E-16
234256	1.7E-08	1.47E-16
707281	3.94E-08	1.24E-16
1679616	8.66E-08	1.89E-16
3418801	3.01E-07	1.97E-16
6250000	4.1E-07	4.25E-16
10556001	4.88E-07	1.6E-16
16777216	1.47E-06	3.58E-16
25411681	1.64E-06	1.36E-16
37015056	2.65E-06	1.28E-16
52200625	4.8E-06	1.19E-16
71639296	6.02E-06	1.3E-16
96059601	4.68E-06	7.87E-16

Based on the sample data recorded in tables 5.8 to 5.9, figure 5.10 graphically compares the Maximum-Max $\Delta\lambda$ and Maximum-Min $\Delta\lambda$ error spread.

Figure 5.10: Increasing Scale of submatrix **B** Test: Max-Min $\Delta\lambda$ Error Spread of a 20, 40 and 60 matrix **H** sizes



For error bound analysis, an increasing scale submatrix **D** is applied to the

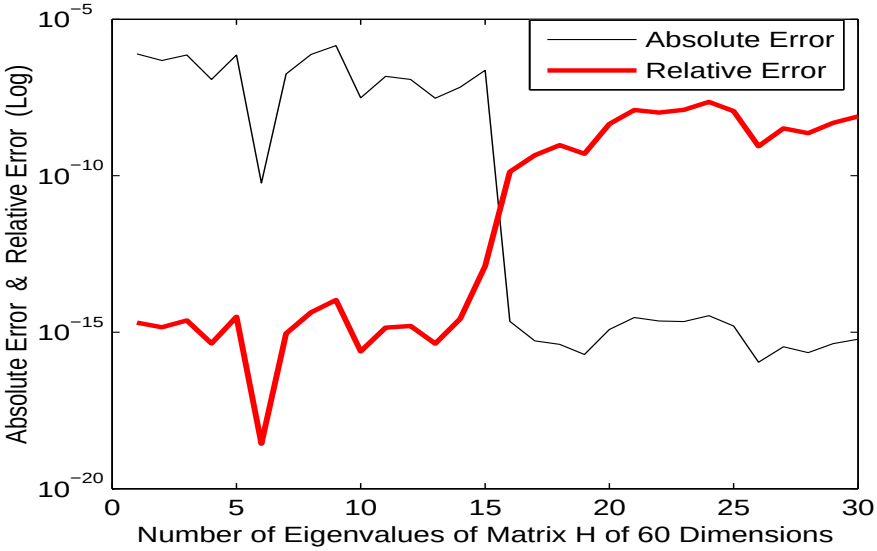
submatrix **B** in a single matrix **H** with dimension 60. From this single scaled matrix **H**, the theoretical error bound is again determined. Due to time constraints, only the maximum scale matrix is used. So the analysis is confined to studying the impacts of the smallest scale matrix **H**. Submatrix **D**(15) takes the form shown below:

$$\mathbf{D(15)} = \begin{pmatrix} 9801 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 9801 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 9801 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \ddots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 9801 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 9801 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 9801 \end{pmatrix}_{\mathbf{30\times 30}} .$$

You will note the 9801*9801=96059601 which is the actual total scale of an entry in **B**

Figure 5.11 sketches out the relative error and absolute error curves for 30 eigenvalues extracted from increasing scaled submatrix **B** with a matrix **H** of size 60. The first eigenvalue at point 1 is the $|\lambda_1|$ and the last point 30 corresponds to $|\lambda_{30}|$ where $|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_{30}|$

Figure 5.11: Relative λ error and absolute λ error from a increasing scaled submatrix $\mathbf{B}$ in a single $\mathbf{H}$ matrix of dimension 60



From this single matrix $\widehat{\mathbf{H}}$ of size 60, Matlab's exact max and min eigenvalues, λ_1 and λ_{30} respectfully, determines the theoretical error bound. These two eigenvalues are:

1. Maximum Eigenvalue $|\lambda_1| \cong 384239005 \cong 3.84 \times 10^8$
2. Minimum Eigenvalue $|\lambda_n| = |\lambda_{30}| \cong .76033 \times 10^{-7}$

From these equivalent singular values of $\widehat{\mathbf{H}}$, the theoretical eigenvalue bounds, eqn (5.1), are:

$$.7753 \times 10^{-6} \cong |\hat{\lambda}_1 - \lambda_1| \ \leq \ 18.2 * \ \epsilon \ |\lambda_{\mathbf{1}}|.$$

In this case, C($m = 60$) is about three factors less than m, C($m = 60$), which is $< .3m$. C($m = 60$) is quite small which makes its a reasonable error bound.

From figure 5.10, the sample Max-Min absolute error does satisfy the theoretical bound. Note, the flipping of magnitudes of the absolute error and relative error in fig. 5.11 is due to the exact eigenvalues decreasing below the value of one. Again, this study indicates that the error bound appears reasonable but it is neither definitive nor comprehensive.

#3 Down Scale B: The down Scale **B** in **H** Test depicts in tables 5.10 to 5.11 the sample data of Max-Max and Mas-Min error spread.

Table 5.10: Max-Min of a down scale **B** of matrix **H** with dim 20 and 40

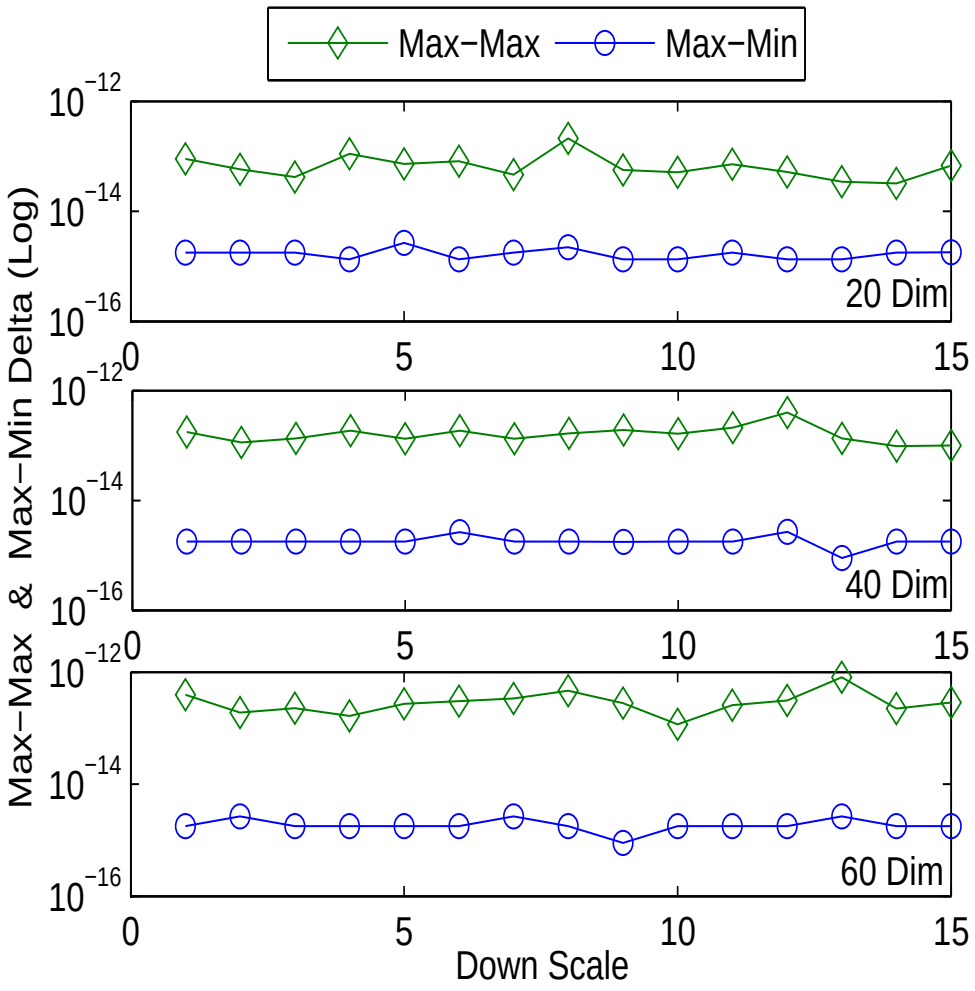
	Matrix 20x20		Matrix 40x40	
Scale	Max-Max $\Delta\lambda$	Max-Min $\Delta\lambda$	Max-Max $\Delta\lambda$	Max-Min $\Delta\lambda$
1	9.01E-14	1.77E-15	1.75E-13	1.78E-15
2	5.82E-14	1.77E-15	1.13E-13	1.78E-15
3	4.22E-14	1.76E-15	1.35E-13	1.78E-15
4	1.11E-13	1.33E-15	1.86E-13	1.78E-15
5	7.28E-14	2.67E-15	1.33E-13	1.77E-15
6	8.08E-14	1.35E-15	1.87E-13	2.66E-15
7	4.66E-14	1.78E-15	1.32E-13	1.77E-15
8	2.11E-13	2.21E-15	1.66E-13	1.77E-15
9	5.64E-14	1.33E-15	1.9E-13	1.77E-15
10	5.11E-14	1.33E-15	1.64E-13	1.78E-15
11	7.15E-14	1.78E-15	2.12E-13	1.78E-15
12	5.2E-14	1.34E-15	3.96E-13	2.67E-15
13	3.46E-14	1.34E-15	1.35E-13	8.96E-16
14	3.2E-14	1.78E-15	9.68E-14	1.78E-15
15	6.79E-14	1.79E-15	1E-13	1.77E-15

Table 5.11: Max-Min of a down scale **B** of matrix **H** with dim 60

	Matrix 60x60	
Scale	Max-Max $\Delta\lambda$	Max-Min $\Delta\lambda$
1	3.94E-13	1.78E-15
2	1.88E-13	2.66E-15
3	2.27E-13	1.78E-15
4	1.64E-13	1.77E-15
5	2.73E-13	1.78E-15
6	3.02E-13	1.77E-15
7	3.39E-13	2.66E-15
8	4.63E-13	1.77E-15
9	2.81E-13	8.95E-16
10	1.16E-13	1.78E-15
11	2.57E-13	1.78E-15
12	3.1E-13	1.78E-15
13	8.12E-13	2.66E-15
14	2.23E-13	1.78E-15
15	2.87E-13	1.77E-15

Based on the sample data recorded in tables 5.10 and 5.11, figure 5.12 graphically compares the Maximum-Max $\Delta\lambda$ and the Maximum-Min $\Delta\lambda$ error spread.

Figure 5.12: Down Scale of submatrix **B** Test: Max-Min $\Delta\lambda$ Error Spread of a 20, 40 and 60 matrix **H** sizes



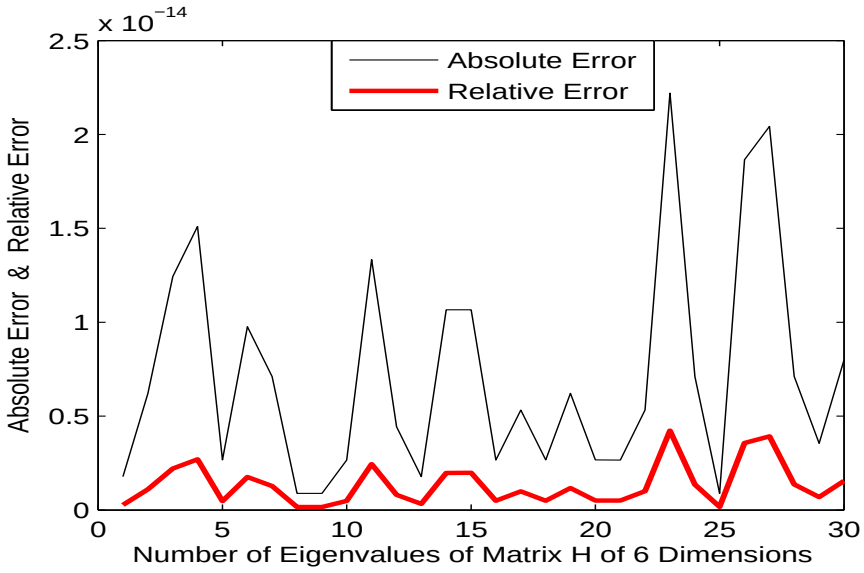
For error bound analysis, a down scale submatrix **D** is applied to the submatrix **B** in a single matrix **H** with dimension 60. From this single scaled matrix **H**, the theoretical error bound is determined. Again, the scale test is not comprehensive

since it is only using one down scale out of fifteen possible. This study analyzes the impacts of the extreme scale submatrix with parameter 'k' = 15. Submatrix $\mathbf{D}(15)$ takes the form shown below:

$$\mathbf{D(15)} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & .18 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & .064 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \ddots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & .00024 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & .00022 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & .000203 \end{pmatrix}_{\mathbf{30\times 30}} .$$

Figure 5.13 sketches out the relative error and absolute error curves for 30 eigenvalues extracted from a down scaled submatrix $\mathbf{B}$ within a matrix $\mathbf{H}$ of size. The first eigenvalue at point 1 is the $|\lambda_1|$ and the last point 30 corresponds to $|\lambda_{30}|$ where $|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_{30}|$

Figure 5.13: Relative λ Error and absolute λ error from a single down scaled matrix $\mathbf{B}$ in $\mathbf{H}$ of dimension 60



From this single matrix $\widehat{\mathbf{H}}$ of size 60, Matlab's exact max and min eigenvalues, λ_1 and λ_{30} respectfully, for determining the theoretical error bound. These two eigenvalues are:

1. Maximum Eigenvalue $|\lambda_1| \cong 6.63$
2. Minimum Eigenvalue $|\lambda_n| = |\lambda_{30}| \cong 5.14$

From these equivalent singular values of $\widehat{\mathbf{H}}$, the theoretical eigenvalue bounds, eqn (5.1), are:

$$.1776 \times 10^{-14} \cong |\hat{\lambda}_1 - \lambda_1| \ \leq \ 2.41 * \epsilon * \ |\boldsymbol{\lambda_1}|.$$

In this case, C($m = 60$) is less than one order of magnitude of m, C($m = 60$) $= 2.41$, which is $< .04m$. C(m) is quite small which makes it a reasonable error

bound.

From figure 5.12, the sample Max-Min absolute error does satisfy the theoretical error bound. However, the sample Max-Max absolute error does not satisfy the error bound. In order to fully bound the sample absolute error data, C(m) must be increased by an 4.7×10. This would mean that C(m)$< .8m$ which is still quite small and the theoretical error bound is still reasonable. Again, the theoretical error bound is based on only one down scale so this is neither definitive nor comprehensive.

#4 Up Scale B: The up Scale **B** in **H** Test depicts in tables 5.12 and 5.13 the sample data of the Max-Max and Max-Min error spread.

Table 5.12: Max-Min of an up scale **B** of matrix **H** with dim 20 and 40

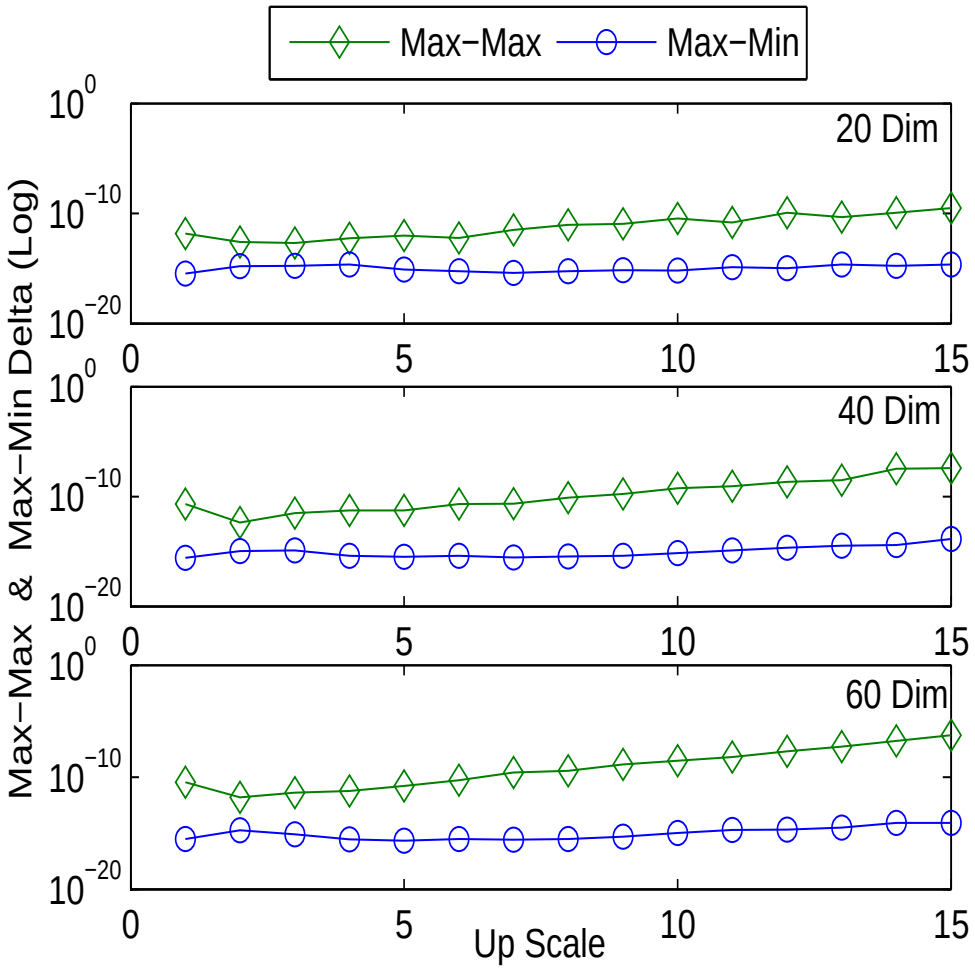
	Matrix 20x20		Matrix 40x40	
Scale	Max-Max $\Delta\lambda$	Max-Min $\Delta\lambda$	Max-Max $\Delta\lambda$	Max-Min $\Delta\lambda$
1	1.44E-12	3.33E-16	2.00E-11	2.57E-16
2	2.63E-13	1.56E-15	4.16E-13	1.11E-15
3	2.06E-13	1.78E-15	3.16E-12	1.22E-15
4	5.40E-13	2.22E-15	5.52E-12	3.89E-16
5	9.43E-13	8.05E-16	5.30E-12	3.31E-16
6	5.97E-13	5.41E-16	2.03E-11	3.89E-16
7	3.28E-12	4.02E-16	2.18E-11	2.91E-16
8	9.18E-12	5.69E-16	7.64E-11	3.59E-16
9	1.15E-11	6.90E-16	1.73E-10	3.90E-16
10	3.50E-11	6.48E-16	5.60E-10	6.88E-16
11	1.55E-11	1.29E-15	8.51E-10	1.22E-15
12	1.16E-10	1.06E-15	2.18E-09	2.23E-15
13	4.55E-11	2.22E-15	2.94E-09	3.33E-15
14	1.16E-10	1.75E-15	3.24E-08	3.91E-15
15	3.00E-10	2.36E-15	3.77E-08	1.34E-14

Table 5.13: Max-Min of an up scale **B** of matrix **H** with dim 60

	Matrix 60x60	
Scale	Max-Max $\Delta\lambda$	Max-Min $\Delta\lambda$
1	2E-11	2.57E-16
2	4.16E-13	1.11E-15
3	3.16E-12	1.22E-15
4	5.52E-12	3.89E-16
5	5.3E-12	3.31E-16
6	2.03E-11	3.89E-16
7	2.18E-11	2.91E-16
8	7.64E-11	3.59E-16
9	1.73E-10	3.9E-16
10	5.6E-10	6.88E-16
11	8.51E-10	1.22E-15
12	2.18E-09	2.23E-15
13	2.94E-09	3.33E-15
14	3.24E-08	3.91E-15
15	3.77E-08	1.34E-14

Based on the sample data recorded in tables 5.12 and 5.13, figure 5.14 graphically compares the Maximum-Max $\Delta\lambda$ and Max-Min $\Delta\lambda$ error spread.

Figure 5.14: Up Scale of submatrix **B** Test: Max-Min $\Delta\lambda$ Error Spread of a 20, 40 and 60 matrix **H** sizes



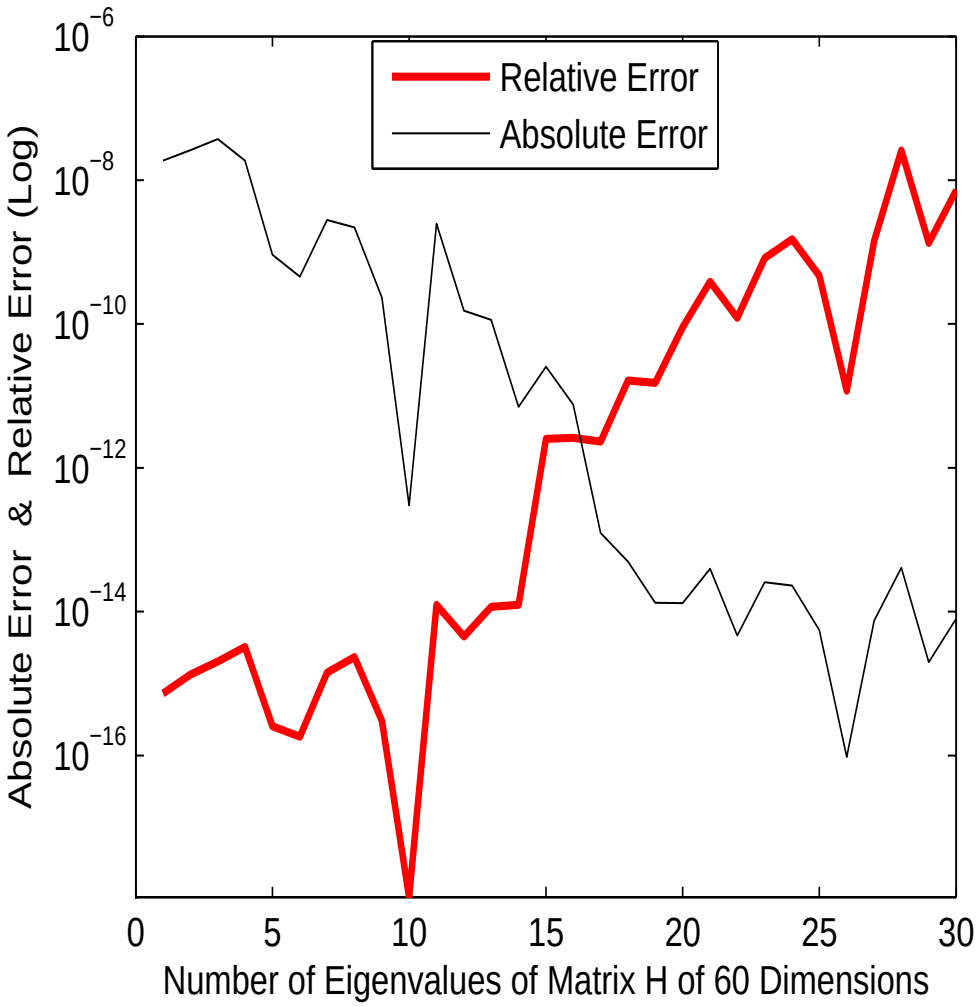
For error bound analysis, an up scale matrix **D** is applied submatrix **B** in a

single matrix **H** of dimension 60. From this single scale matrix **H**, the theoretical error bound is determined. Only one scale test was used, therefore the test is not comprehensive. Submatrix **D**(15) takes the form shown below:

$$\mathbf{D(15)} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 5.7 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 15.6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \ddots & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 4148.5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4528.9 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4929.5 \end{pmatrix}_{\mathbf{30\times 30}} .$$

Figure 5.15, it sketches out the relative error and absolute error curves for 30 eigenvalues extracted from an up scaled submatrix **B** within a matrix **H** of size 60. The first eigenvalue at point 1 is the λ_1 and the last point 30 corresponds to $|\lambda_{30}|$ where $|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_{30}|$

Figure 5.15: Relative λ Error and absolute λ error from a single up scaled $\mathbf{B}$ of matrix $\mathbf{H}$



From this single matrix $\widehat{\mathbf{H}}$ of size 60, Matlab's extracts the exact max and min eigenvalues, λ_1 and λ_{30} respectfully, for determining the theoretical error bound. These two eigenvalues are:

1. Maximum Eigenvalue $|\lambda_1| = 25789542 = 2.58 \times 10^7$

2. Minimum Eigenvalue $|\lambda_n| = |\lambda_{30}| = .1106 \times 10^{-5}$

From these equivalent singular values of $\widehat{\mathbf{H}}$, the theoretical eigenvalue bounds, eqn (5.1), are:

$$.187 \times 10^{-7} \cong |\hat{\lambda}_1 - \lambda_1| \ \leq \ 6.53 * \ \epsilon * \ |\lambda_{\mathbf{1}}|.$$

In this case, C($m = 60$) is less than one orders of magnitude of m, C($m = 60$) = 6.53, which is $< .11m$. C(m) is quite small which makes it a reasonable error bound.

From figure 5.14, both the sample Max-Min and Max-Max absolute errors satisfy the theoretical error bound. Again, this is neither definitive nor comprehensive; but, it does indicate that the algorithm is reasonably bounded.

To recap the entire numerical test, matrix **H** was tugged and pulled from its base uniform entry structure. A random matrix **H** was created and the theoretical error bounds from a test matrix were determined and compared to The Methods errors. The test then progressed to an increasing and a decreasing scale over the entire matrix **H** with the same theoretical comparison. Finally, using four scaling processes, only the submatrix **B** within matrix **H** was scaled. After scaling the submatrix, the theoretical error bounds derived from a scaled test matrix were compared to The Methods error spread. Again, the test matrix was based on the same scaled structure that The Method algorithm was applied to. All test matrix were of size 60 and only one test matrix was created for the uniform test and each scaled tests. Below is a summary of how The Method performed under these tests.

Table 5.14: Summary of Numerical Tests

Scale	Max-Min $\Delta\lambda$	C($m = 60$)$< M$
Uniform	Bdd	C($m = 60$)$< 3m$
Dec. Scale	Bdd	C($m = 60$)$< 1.83 \times 10^5 m$
Inc.Scale	Bdd	C($m = 60$)$< 3m$
Dec. **B**	Bdd	C($m = 60$)$< 10m$
Inc. **B**	Bdd	C($m = 60$)$< .3m$
Down **B**	Bdd	C($m = 60$)$< .8m$
Up **B**	Bdd	C($m = 60$)$< .11m$

where Bdd means Bounded and M is just a constant.

In this analysis, only one scaled matrix **H** with dimension size 60 was created for each test for determining the theoretical eigenvalue error bounds for the algorithm. The matrix's theoretical bound was then compared to the sample absolute error data from the corresponding numerical test. In all numerical test cases except for the decreasing scale matrix **H** Test, the theoretical bound error bounds for C(m) were reasonable in size and did not exceed $10m$. However, for the decreasing Scale Matrix **H** Test, the error bounds for C(m) were not reasonable in size. C(m) $< 1.83 \times 10^5 m$ which is very large. A possible answer for this large error bound may be due to severe cancellation from such small entries produced from the scale.

Since this test relied only on one matrix for each test in determining the theoretical bounds, the test is not statistically definitive nor comprehensive. To be definitive , estimates for C(m) would be required for each test and then statistically averaged. In addition, for each dimension and scale, a C(m) would have to be determined and statistical averages determined. Comparisons of this theoretical bound to the error spread would be more comprehensive and definitive.

Therefore, based on the numerical test results, The Method's eigenvalue error spread behaves reasonable well.

Chapter 6

Conclusion and Findings

The Method's algorithm was developed to leverage both the skew-symmetric properties and the zero (2,2) block in a Gyroscopic QEP in order to speed up the eigenvalue finding process as compared to the symmetric QR algorithm. Flop counts verify that determining eigenvalues is approximately $\frac{8n^3}{3}$ faster than the symmetric QR algorithm. However, to get a more comprehensive comparison on flop count efficiency, future studies would be required to determine the cost in creating the transformation matrix $\mathbf{Q}$ as compared to the symmetric QR's $\mathbf{Q}$'s cost. For accuracy, although neither definitive nor comprehensive, the Maximum and Minimum absolute errors satisfy the theoretical error bounds. It is not a definitive conclusion since only one theoretical eigenvalue error bound sample was used. To become statistically definitive and comprehensive, the numerical test should be repeated with the provision that theoretical error bound's statistical average is extracted from each test sample population both in size of matrix and scale applied. With the sample average theoretical error, we can then compare whether the absolute error spread truly was reasonably bounded. The only anomaly detected in the error bound was when the extreme small scaler matrix $\mathbf{D}$ was applied. Since both Matlab's eig() and The

Method was used in finding the eigenvalues. The eigenvalues can be expected to be very small for both numerical process. This study suspects that when applying the absolute error with these extremely small numbers, severe cancellations occurred which caused unreasonable theoretical error bounds. Besides this anomaly, all other scaling tests had reasonable error bounds where the bounds were with one order of magnitude of the matrix dimension equaling to $m = 60$. Because of reasonable error bounds and cheaper eigenvalue finding algorithm costs, The Method appears to be equivalent in accuracy to the QR method but cheaper in cost than the QR method.

Future work in analyzing The Method is first to ensure that for each test sample both in size of matrix and type of scale, the theoretical error bound's statistical average is extracted from the test sample population. Second, the algorithm developed for The Method listed in Appendix B must be improved. In appendix B, the algorithm did leverage most of the symmetry advantages required to reduce the flop costs. However, the algorithm did not leverage 100 percent of the flop costs outlined in chapter five's flop analysis. Once this symmetry is fully incorporated in the algorithm, then CPU times can be analyzed for cost-efficiency comparison between Matlab's eig() and The Method. CPU time comparison would then be another metric in comparing the efficiency of The Method to other eigenvalue finding algorithms. Third, cost analysis for saving the transformation matrices in The Method was not done. In order to fully compare cost-efficiency of The Method to the QR or other methods, this analysis should be done for a more all inclusive approach in cost-efficiency comparison. Fourth, determine a matrix $\mathbf{H}$ with known eigenvalues and determine the absolute error and relative error of The Method's eigenvalues to the known eigenvalues. Finally, perturbation analysis is the next major step and really is the primary reason for analyzing this algorithm, The Method.

Since The Method preserves the QEP structure through most of the entire process until the SVD is applied, perturbation analysis can be analyzed for not only the whole matrix but within the submatrices as well. As each transformation iteration is applied, due to round off errors, $\widehat{\widetilde{\mathbf{H}}}$ is not exact. This means that $\widehat{\widetilde{\mathbf{H}}} = \widehat{\mathbf{H}} + \mathbf{E}$ where $\mathbf{G}_{ij}^T \mathbf{H} \mathbf{G}_{ij} = \widehat{\widetilde{\mathbf{H}}} = \widehat{\mathbf{H}} + \mathbf{E}$ and $\widehat{\mathbf{H}}$ is the exact transformation and $\mathbf{E}$ is the error matrix. When using The Method, then the error matrix $\mathbf{E}$ has the same QEP structure as matrix $\mathbf{H}$. With the structure preserved in the error matrix, it may be possible to determine a backward error bound of the calculated eigenvalue error[10].

Appendix A

Cholesky-like factorization

A.0.1 Cholesky factorization $\mathbf{A = L\ L^T}$

Using Trefethen's proof in [5], let a symmetric positive definite matrix be

$$\mathbf{A} = \begin{pmatrix} a_{11} & w^T \\ w & \mathbf{K} \end{pmatrix} \text{ with the dimensions } R^{m \times m} \text{ and } a_{11} > 0$$

where $\alpha = \sqrt{a_{11}}$ and w is a vector of length m-1 and $\mathbf{K}$ is a square matrix with dimensions (m-1)×(m-1). Then $\mathbf{A}$ is factorized in the following manner:

$$\mathbf{A} = \begin{pmatrix} \alpha & 0 \\ \frac{w}{\alpha} & \mathbf{I} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & \mathbf{K} - \frac{ww^T}{a_{11}} \end{pmatrix} \begin{pmatrix} \alpha & \frac{w^T}{\alpha} \\ 0 & \mathbf{I} \end{pmatrix} = \mathbf{L_1 A_1 L_1^T}$$

The same process is repeated with $\mathbf{A}_1 = \mathbf{L_2 A_2 L_2^T}$ such that $\mathbf{A} = \left(\mathbf{L_1\ L_2\ A_2\ L_2^T}\right)\ \mathbf{L_1^T}$. This iteration continues until $\mathbf{A = L_1\ L_2\ \cdots\ L_m\ L_m^T\ \cdots\ L_2^T\ L_1^T\ =\ L\ L^T}$

A.0.2 Cholesky-Like factorization $\mathbf{A = R\ R^T}$

We repeat the proof of [5], except the process begins the factorization in reverse direction.

$$\mathbf{A} = \begin{pmatrix} \mathbf{K} & w \\ w^T & a_{mm} \end{pmatrix} \text{ with the dimensions } R^{m \times m} \text{ and } a_{mm} > 0$$

where $\alpha = \sqrt{a_{mm}}$ and w is a vector of length m-1 and $\mathbf{K}$ is a square matrix with dimensions (m-1)×(m-1). Then is factorized in the following manner:

$$\mathbf{A} = \begin{pmatrix} \mathbf{I} & \frac{w}{\alpha} \\ 0 & \alpha \end{pmatrix} \begin{pmatrix} \mathbf{K} - \frac{ww^T}{a_{11}} & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \mathbf{I} & 0 \\ \frac{w^T}{\alpha} & \alpha \end{pmatrix} = \mathbf{R_1}\ \mathbf{A}_1\ \mathbf{R_1^T}$$

Again, the same process is repeated with $\mathbf{A}_1 = \mathbf{R_2 A_2 R_2^T}$ such that $\mathbf{A} = \mathbf{R_1}\ \left(\mathbf{R_2}\ \mathbf{A_2}\ \mathbf{R_2^T}\right)\ \mathbf{R_1^T}$. This iteration continues until $\mathbf{A} = \mathbf{R_1}\ \mathbf{R_2}\ \cdots\ \mathbf{R_m}\ \mathbf{R_m^T}\ \cdots\ \mathbf{R_2^T}\ \mathbf{R_1^T}\ =\ \mathbf{R}\ \mathbf{R^T}$

Appendix B

Program Codes

B.1 Program Code for the method

B.1.1 the method main code

```
%function [T, Eigenvalues] = MethodI(H)
%%Similarity and congruence transformation of a matrix H to a real schur
%%form. In Final Form 13 June 08
%
%% Wade Rush KU Master Thesis 15 March 06
%%                      edited 28 Feb 08
%%                      edited 2 March 08  Runs to real Schurr form
%%                      edited 18 March 08
%%                      edited 29 May 08 adjusted to ensure actual real
%                                %Schur Form
%%                      13 June  eliminated G, GUB and PSVD call functions
%%                      and made them sub functions in methodI
```

```
%
%%  Input:   H  -is a skew-symmetric matrix H
%%  Output:  T - is the factorized matrix H into the real Schur form
%%  Output:  Eigenvalues are the positive imaginary eigenvalues of
%%           matrix H in descending order
%
%% Purpose:  Perform a similarity and congruence transformation of a random
%% generated skew symmetric matrix of 2n-by-2n in the form of
%% H = [-B -L,LT O]
%%
%
%
%% Definition of variables
%%
%% N =2*n
%%
%% H =  [-B -L, LT O] a skew symmetric matrix N-by-N
%%
%% B =  skew symmetric submatrix  n-by-n  of H
%%
%% L =  uppertriangle submatrix n-by-n of H
%%
%% LT = the transpose of L
%%
%% O = zero n-by-n submatrix of H
%
```

```
%% BL = [-B -L]
%%
%% Gij = the 2-by-2 rotational matrix
%
%% G(x1,x2) is a subfunction Givens rotation shifting the mass from x1 to x2
%% such that G(x1,x2) creates a Givens matrix
%      %S.T. G(x1,x2)*[x1;x2] = [0; xx2]
%
%% GUB(x1,x2) is a subfunction Givens rotation shifting the mass from x2 to x1
%% such that GUB(x1,x2) creates a Givens matrix
%s.t. GUB(x1,x2)*[x1; x2] = [xx1; 0]
%
%%-------------------------------------------------------------------------
%% Step 1:  Setting up the skew symmetric matrix then focusing on the upper
%% blocks [-B, -L]
%
%[M,N]=size(H);    % dimensions row and column of H
%
%n = N/2;
%
%
%BL = H(1:n, :);     % Maximizing sparsity and nature of skew matrix, hence
%                       %Only need to deal with [-B -L] matrix. Note
%                       %replaced FF
%
%for i = 1:n-1   %( Loop 1 for ) (i = Row counter)   Have to use -2 in
```

```
%                       % order to not cause the Gij to bleed into
%                       % L matrix. It defines the
%                       % row that the method must operate on.
%
%%
%% Begin iteration to reduce matrix so -B -> 0 and L -> upper - bidiagonal
%
%%Step 2:   Eliminates B matrix on row i (minus last column of B on row i)
%
%    for j = i+1:n-1 %( Loop 2 for ) (j = column counter) eliminate B up
%                    % to column n
%        %BL(i,j);   % Checker to see if inputing correct items
%
%        %BL(i,j+1); % Checker to see if inputing correct items
%
%                                    % subfunction G (a rotation  matrix)
%        Gij = G(BL(i,j),BL(i,j+1)); % Rotation matrix placing mass of x1
%                                    %  into x2 row and column i
%
%%Note: Gij*H*Gij' similarity transformation is equivalent in the
%%next two lines
%
%        BL(j:j+1,:) = Gij*BL(j:j+1,:);  % Acts on two rows
%
%        BL(:,j:j+1) = BL(:,j:j+1)*Gij'; % Acts on two columns
%
```

```
%% Step 2a:  Cleaning up contaminates created in L from above similarity
%% transformation
%
%        Gij = G(BL(j+1,n+j),BL(j+1,n+j+1));
%
%        BL(:,n+j:n+j+1) = BL(:,n+j:n+j+1)*Gij';
%
%%       if i ==6;
%%          disp('shifted one'), disp(i)
%%          BL
%%       end
%%       if i==1 || i==3
%%          i
%%       end
%    end  %(Loop #2 for ) which eliminates row i's last column entry B(i,n)
%
%%          disp('On Row '),i
%%
%%          disp('What BL row looks like except last column n'),BL
%
%        %  Between loop #2 and beginning of loop#3, it shifts last
%        %  row i entry
%        %  in Matrix B, n column,  to N column in row i of the
%        %  Upper Triangle
%
%
```

```
%        Gij = G(BL(i,n),BL(i,N));  % Eliminates last B(i,n) and
%                                   % sends the mass of info to L(i,n) or
%                                   % now UT(i,n)
%
%       % Problem here because we chopped off the bottom half of the
%       %matrix A
%
%       % Hence, in order to symmeterize B without multiplying the whole
%       %matrix with Gij on the left side of the BL matrix we substitute.
%
%        BLnN = BL(n,N);  %append 18 March 08
%
%        BL(i:n,[n N]) =  BL(i:n,[n N])*Gij';
%
%        BL(n,1:n)= -1*BL(1:n,n)';  % Substitutes thus avoiding apply Gij
%                                   % on left side.
%        BL(n,N) = BLnN;  %append 18 March 08
%
%        BL(n,n)=0;  % If you don't do this, the ij entries are not
%                    % perserved since we
%                    % are not actually applying the left Givens rotation.
%%       disp('Shifted column n to UT N')
%%        BL;     % Just a checker
%
%    %Step 3:  Reduce matrix UT's row i to an upper Biadiagonal form.
%
```

```
%%     ii = 0 ; % Just a count checker
%
%       for jj=0:n-i-2 %Loop 3
%       %Loop 3 then takes row i of the upper triangle and shifts the
%       %information entries into upper bidiagonal form.
%       % along with clean up.
%
%       BL(:,N-jj-1:N-jj) = BL(:,N-jj-1:N-jj)*GUB(BL(i,N-jj-1),BL(i,N-jj))';
%       %Shift rows toward Upper-bidiagonal
%           % Calls function GUB which is an M-File givens matrix.
%
%           % Clean up operation  GUB*H*GUB'
%           %
%           % disp(' what does BL(n-jj-1:n-jj, look like')
%           % BL(n-jj-1:n-jj,:);
%           % disp('left givens')
%           gub = GUB(BL(n-jj-1,N-jj-1),BL(n-jj,N-jj-1));
%           BL(n-jj-1:n-jj,:) = gub*BL(n-jj-1:n-jj,:);
%
%%          disp('right givens')
%
%           BL(:,n-jj-1:n-jj) = BL(:,n-jj-1:n-jj)*gub';
%
%
%%            disp('iteration of upper-diagonal')
%%            disp('for row'), i
```

```
%%            jj
%%            BL;
%
%       end %(Loop #3 For )
%
%end %(Loop #1 For ) Primary loop
%
%   % step 4:  Eliminate the last bit of B (then done)
%
%   Gij = G(BL(n-1,n),BL(n-1,N));
%
%   BL(:,[n,N])=BL(:,[n,N])*Gij;
%
%   BL(n,n) =0;  % By passes having to multiply all of matrix A
%
%   BL(n,n-1)=0;  % By passes having to multiply all of matrix A
%
%   [E_values, T] = PSVD(BL); % output
%
%   p = length(E_values);
%
%   Eigenvalues = zeros(p,1);  % initializing dummy vector
%
%   for i = 1:p  % reordering from largest to smallest evalue
%       Eigenvalues(i) = E_values(p+1-i);
%   end
```

```
%
%end % end of MethodI
%
%function Gij = G(x1,x2)
%%Gij(theta) rotational 2-by-2 orthogonal rotations shift mass from
%% right to left for row and shifts mass from top to bottom in columne.
%
%% Wade Rush  Math master thesis Modified 25 Feb 08
%                               % Modified 18 March 08
%%                              % Modified 22 May 08
%
%% Definition of variables
%%
%% x1 =  some entry in a matrix.
%% x2 =  some other entry in a matrix
%% x1 and x2 displayed as a row vector [x1, x2] or a column vector [x1; x2],
%% in that order.
%% theta = The angle required to rotate and eliminate x1 and place the mass
%%           into x2 entry.
%% Gij = [cos(theta) , sin(theta); -sin(theta) , cos(theta)];
%
%
%
%%  Gij(2,3)*[2;3] = [0; 3.6056]  Column vector and [2;3]'*Gij(2,3)'=
%%     [0; 3.6056]' row vector as an
%%  example
```

```
% %
%%-------------------------------------------------------------------------
%%
%% if x2 == 0 ||  x1 == 0; %if x1 is already zero then the operation is
%% not required
%%     Gij =[1, 0; 0, 1];  %  I think a semi-colon is needed to eliminate
%%a break error msg
%% % elseif x2 == 0
%% %     theta = 0;
%if x1 == 0;
%    Gij = [1, 0; 0, 1];
%elseif x2== 0;
%    Gij = [0 1;-1 0]; % Because theta is 90 degrees
%else theta = acot(-x2/x1);  %App 18 March: Original was: theta = -acot(x2/x1)
%    Gij = [cos(theta) , sin(theta); -sin(theta) , cos(theta)];
%end
%
%end  % G function
%
%function Gij = GUB(x1,x2)
%%GUB(theta) rotational 2-by-2 orthogonal rotations which shifts mass of a
%%  two entry row from
%%  right to left  or for column from bottom to top.  It is focused at a
%%  Givens rotation to transform
%% H rows into upper bidiagonal(GUB) and special cleanups .
%
```

```
%% Wade Rush  Math master thesis Modified  1 Mar 08
%%                               Modified 22 May 08
%%
%% Definition of variables
%%
%% [x1; x2] or [x1,x2] are two entry in a matrix which form a 2-column
%%       and 2-row vector respectfully
%%
%% theta = The angle required to rotate and eliminate x2 and place the
%%           mass into x1 entry.
%% GUB = [cos(theta), sin(theta) ; -sin(theta), cos(theta)];% Same
%         % structure as m-file G
%
%%  [2 3]*GUB(2,3)' = [3.6056 0]  Row vector as an example
%
%%  GUB(2,3)*[2 3]' = [3.6056  0]'
% %
%%-------------------------------------------------------------------------
%
%if x2 == 0;
%    Gij = [1, 0; 0, 1];
%elseif x1== 0;
%    Gij = [0 1;-1 0];
%else theta = acot(x1/x2);
%    Gij = [cos(theta), sin(theta) ; -sin(theta), cos(theta)];
%end
```

```
%
%end % subfunction GUB
%
%function [Eigenvalues, T] = PSVD(UBD)
%% Diagonalize using SVD and Permutation of the input UBD matrix where
%%   UBD = [0  Bi-diag].
%% Output is eigenvalues of matrix H = [0  Bi-diag; -Bi-diag  0] and the
%% matrix T is the real Schur form transformation of matrix H
%
%%@Author Wade Rush
%% created   March 08
%% modified 22 May 08 updated comments and updated  by sorting d vector
%% modified 28 May 08
%
%%--variable definitions----------------
%% UBD = [0  Bi-diag]  where Bi-diag sub-matrix isupper bi-diagonal.
%% UBD is an n-by-N matrix where N =2*n
%
%%  0 sub-matrix with dimenison n-by-n
%%  Bi-diag sub-matrix with dimensions n-by-n
%
%[n,N]=size(UBD);
%
%%n; % row
%
%%N; % column
```

```
%
%[U ,E,V] = svd(UBD(:,n+1:N));%  SVD operation on upperdiagonal of UBD
%                               %  to find E
%
%T = zeros(N,N);  % Initialize a T matrix of dimensions N-by-N
%
%T(1:n,n+1:N) = E; %  Building the upper E in the transformed UBD matrix
%
%T(n+1:N,1:n)=-E;  %  Building the lower -E in The transformed UBD matrix
%
%%T;  % T is the 2nd output of this function and T is in
%%the real schur form: T = [0 E; -E 0] where E is the eigenvectors + or -
%% without the imaginary.
%%P=eye(N); % Building the permutaton matrix.  If I used permutations P, it
%% would organize
%% transform the T matrix in to box intries of 2-by-2 such that [0 a; a o]
%% which implies that evalue is + or - i*a where "i" is imaginary.
%
%%P = P(:,[2*(1:n)-1,2*(2:n)]);
%
%%P';
%
%%D = Pi*T*P
%
%%D = P*T*Pi
%
```

```
%
%% Instead of doing the permutation, we can cut to the chase and
%% extract it directly from the E sub-matrix.
%
%d = diag(T(:,N/2+1:N));  %  Represents all the positive evalues
%                          %  not the negative evalues %modified 22 May 08
%
%Eigenvalues = sort(d);  %  Is the first output of this function.
%                        %  Modified 22 may 08
%
%end % end of PSVD subfunction
%
%
```

B.1.2 Subroutines to MethodI: Build skew-symmetric matrix

```
%function H = SKSM(n, flag)
%% Develop a random skew symmetric matrix in the form [-B,-R;R^T 0] o
%
%%input:  2*n  - representing dimensions of B sub-matrix as well as R
%               %   sub-matrix dimensions.
%%input:  flag - switch operation
%                % flag =1 Then execute one method in making B matrix
%                          % but it is not that sophisticated
%                          % H is 4*n dimensions
%                % flag =2 Then execute better method in making B matrix
%                          % more sophisticated and comprehensive
```

```
%                          % H is 4*n dimensions
%
%% Define variables
%% A =  4n-by-4n skew symmetric matrics in the form [-B,-R;R^T 0]
%% 2*n =  is the input for the size of submatrix blocks, B, R and 0.
%% matrix
%% B =  2*n-by-2*n skew symmetric matrice
%% R =  2*n-by-2*n upper triangle matrice
%% 0 =  2*n-by-2*n null matrice
%% R^T = R transpose
%
%% Wade Rush  In KU mathematics master program
%% Appended  18 March 08  Increased B1 by changing the coefficient of
%%random matrix to 10
%% Appended  1 June 08   added switch, clarified with comments, added
%% another method to build a random H matrix
%
%
%switch flag
%
%    case 1
% nn= 2*n;
%
% B1 = 10*rand(nn,nn);  %  Generates a random generated 2n-by-2n matrix
%
% B3 = B1 - B1';  %  Generates a skew-symmetric 2n-by-2n matrix
```

```
%
% zero_matrix = zeros(n,n);
%
% B3(n+1:nn, n+1:nn) = zero_matrix; % replace with section with
%                                     % 0 sub-matrix lower right quadrant
%
% %  zero out upper right quadrants lower triangle thus creating
% % L sub-matrix
%
% for j = 1 : n
%             for i = n+1 : nn
%                 if j > i-n
%                    B3(j,i)=0;
%                 end
%             end
% end
%
% %  zero out lower right quadrants upper triangle thus creating
%    %L^T sub-matrix
%
% for j = n+1: nn
%             for i = 1 : n
%                 if j < i + n
%                 B3(j,i)=0;
%                 end
%             end
```

```
% end
% H = B3;
%
%    case 2
%    s = n;  % pulled this entire case 2 from another program so decide
%                %   to continue to use the variable "s"
%
%    N = 2*s;  %Establishes the dimensions of submatrix B
%
% % Create sub-matrix B
%
%        % Create B_Initial matrix
%              Is = eye(s);    %  Identity matrix of s-by-s dimensions
%
%              Os = zeros(s);  %  zero Matrix of s-by-s dimensions
%
%              Og = zeros(N);  %  Create a zero matrix of 2*s-by-2*s
%                              %  dimensions
%
%              [Q,R] = qr(rand(N));  % QR factor to extract the
%              %orthogonal Q matrix of dimensions 2*s-by-2*s
%              % Changed from randn(N) to rand(N) 19 october
%
%              B_Initial = [Os Is;-Is Os];  % B initial with known
%              %structure:Doesn't do us much good but still structure
%              %is known and consistent
```

```
%
%
%               B = Q'*B_Initial*Q;     % Scramble B a bit with structure
%               % and evalue preserving unity transformations.
%
% %  Create a positive definite sub-matrix CC
%        %  Note CC = rand(n, n) + (n - 1)*eye( n ).
% %  Note: we can test whether a matrix is positive definite by the
% %  following:
% %chol( rand( 10, 10 ) ); ??? Error using ==> chol, Matrix must be
% % positive definite.
%
%                 % This was found on the web page:
% %http://www.ece.uwaterloo.ca/~ece204/TheBook/04LinearAlgebra
% %             /posdef/complete.html;
% %Univ of Waterloo, Numerical Methods for Electrical and Computer
% %        Engineers.
% % Department of Electrical and Computer Engineering
% % University of Waterloo
% % 200 University Avenue West
% % Waterloo, Ontario, Canada N2L 3G1
% % 519 888 4567
%
%            CC = rand(N, N) + (N - 1)*eye( N );
%
% %  Create R, upper triangle submatrix, from chol(CC)
```

```
%
%             R = chol(CC);
%
% %  Create H = [-B  -R; R^T 0]
%
%             H = [-B  -R; R' 0g];
%
%end
%
```

B.2 Numerical Tests

B.2.1 Uniform Test

```
%function  [StdMin_v, StdMax_v, MeanMin_v, MeanMax_v,Dim_v, Max_Max_v,
% Max_Min_v ] = UMatrixTest(Max_Dim, m)  %
%
%% Generate a uniform QEP matrix [-B -R;R^T 0] then compares eig() and
%% methodI in terms of maxmax maxmin evalue differences iterated from
%% diminsions of 4 to multiples of 4 dim to max-dim in matrix size and
%%we do this m times per matrix dimension.
%
%
%% Created by Wade Rush
%% 5 June 2008
%% 16 June 08  Added the statistical analysis
%
```

```
%% Input:  Max_Dim -describes max dimension is the dimension of the
%%         largest matrix created.  Max_Dim must be a even numbers
%          %which is multiples of 4.
%
%% Input:  m  -defines how many iterations per matrix dimensions.
%% Output: Dim_v  -contains the dimensions of every new matrix
%            %dimension in vector form
%% Output: Max_Max_v  -Maximum of maximum change in eigenvalue in vector
%                     % form where entries correspond to size (dimension)
%                     %of matrix
%% Output: Max_Min_v  -Maximum of minimum change in in eigenvalue in vector
%             % form where entries correspond to size (dimension)of matrix
%%
%%Pseudo code
%tic
%if mod(Max_Dim,2)==0
%    disp('Initiating test run');
%
%    NN= Max_Dim/4;
%    Vector = 1:NN;
%    %g=0;
%
%    % Initializing table vectors
%    Dim_v = 4*Vector'; % Actual value
%    Max_Max_v  = zeros(NN, 1); % Collects the max of max delta lambda for
%                               % each matrix dim
```

```
%    Max_Min_v  = zeros(NN, 1); % Collects the max of max delta lambda for
%                               % each matrix dim
%    MeanMax_v = zeros(NN,1);  % Collects the mean of the max delta lambda
%                               % for each matrix dim
%    MeanMin_v = zeros(NN,1);  % Collects the mean of the min delta lambda
%                               %for each matrix dim
%    StdMax_v = zeros(NN,1);  % Collects the std of the max delta lambda
%                              %for each matrix dim
%    StdMin_v = zeros(NN,1);  % Collects the std of the min delta lambda
%                              %for each matrix dim
%    Max_v = zeros(m,1); % Collects all the max delta lambdas per matrix
%                        %  iteration
%    Min_v = zeros(m,1); % Collects all the min delta lambdas per matrix
%                        % iteration
%
%    count = 0;
%
%
%%I.  Design a H matrix with a specific structure
%       % create subfunction create matrix: sub functio H-Marix does this
%
%
% %  II.  Run a for loop for n times " for increasing size of H-matrix".
%%  However, there is a limit since eig(vpa(H)) is extremely slow.
%%
%%  Tic Toc timing eig() matlab ftcn   toc  100*toc sec In minutes
```

```
%% Elapsed time for diagonalizing a 4 dim matrix =  0.056987  5   .1 min
%% Elapsed time for diagonalizing a 8 dim matrix =  0.20102  17  .29 min
%% Elapsed time for diagonalizing a 12 dim matrix = 0.39731  39. .65 min
%% Elapsed time for diagonalizing a 16 dim matrix = 0.77258  75. 1.25 min
%% Elapsed time for diagonalizing a 20 dim matrix = 1.2978   130 2.2 min
%% Elapsed time for diagonalizing a 24 dim matrix = 2.0827   208 3.5 min
%% Elapsed time for diagonalizing a 28 dim matrix = 3.271    327 5.4 min
%% Elapsed time for diagonalizing a 32 dim matrix = 4.4054   441 7.4 min
%% Elapsed time for diagonalizing a 36 dim matrix = 6.0037   600  10 min
%% Elapsed time for diagonalizing a 40 dim matrix = 8.0308   803  13 min
%% Elapsed time for diagonalizing a 44 dim matrix = 11.0882  1109 18 min
%% Elapsed time for diagonalizing a 48 dim matrix = 13.3909  1339 22 min
%% Elapsed time for diagonalizing a 52 dim matrix = 16.7709 1677.9 28 min
%% Elapsed time for diagonalizing a 60 dim matrix = 25.3473  2535  42 min
%
%
%       for s=1:NN   % Each s increases H matrix dimension 4*s
%           for i=1:m  % Number of iterations for each H matrix dimensions
%               %s
%               % i  counter
%               [B_I,Og_,N] =Initial_B(s);
%               H = H_Matrix(B_I,Og_,N);
%
%               %  Determine evalues using methodI
%               [T, Eigenvalues] = MethodI(H);
%
```

```
%               %  Determine evalues using Matlab eig(H) QR iteration and
%               % methodI's iteration
%
%               [Exact_Evalues, p] = QRSort(H); % List exact evalues:
%               % QRSort vpa's H and sorts positive imaginary evalues in
%               %descending order
%
%
%               % Determine difference between each corresponding E-values.
%
%               diff_lambda = zeros(p,1); % Initialize diff_Lambda vector
%
%               for j=1:p   %  Determine diffence between exact and methodI
%                           %  evalues
%                 diff_lambda(j) = double(abs(Eigenvalues(j)-Exact_Evalues(j)));
%               end
%
%               Max_v(i) = max(diff_lambda);
%               Min_v(i) = min(diff_lambda);
%
%
%           end
%              count = count + 1;
%              MeanMax_v(count) = mean(Max_v);
%              MeanMin_v(count) = mean(Min_v);
%
```

```
%               StdMax_v(count) = std(Max_v);
%               StdMin_v(count) = std(Min_v);
%
%               Max_Max_v(count)  = max(Max_v);
%               Max_Min_v(count)  = max(Min_v);
%
%       end
%
%  %EstEvals_v= Exact_Evalues
%  %EVals_v=Eigenvalues
%
%
%%Pseudo Code:
%
%%  I   a.  For each n dimensional H-matrix, we create this matrix m
%%      times by running a for loop.  The size of  m could perhaps be
%%       as large as m = 100"
%
%%      b.  Each H-matrix created, run QRsort() and methodI() and
%%       corresponding e-values from each other.
%
%%      c.  Determine the max change and min change in this iteration and
%%      store this into Max_vector and Min_vector which should be size m
%
%%      d.  Sort Max_vector and Min_vector and determining Max-max and
%%      Max-min and store values into Max_Max_v and Max_Min_v which should
```

```
%%      be size n
%
%%      e.  Store into Dim_v the size of the matrix for each iteration.
%
%%      f.  Transfer Dim_v, Max_max_v and Max_Min_v into cell array called
%%      UTable where U is uniform matrix distribution.
%
%%      g.  XLMwrite UTable to excel and then transfer data to Latex.
%
%     disp('Finishing test run');
%
%else
%    disp('Input was not an even number')
%end
%
%disp(['Elapsed time for running test with number of matrix iterations =
%' num2str(NN) ' is ' num2str(toc) ]);
%end % MatrixTest
%%-------------------------------------------------------------------------
%function  [B_Initial,Og,N] = Initial_B(s)
%% Input: s -is the 1/2 the size of B matrix or 1/4 size of H matrix
%% Output:  B_Initial -is the Inital B matrix condition but not final stage
%
%     N = 2*s;  %Establishes the dimensions of submatrix B
%
%  % Create B_Initial matrix
```

```
%       Is = eye(s);    %  Identity matrix of s-by-s dimensions
%
%       Os = zeros(s);  %  zero Matrix of s-by-s dimensions
%
%       Og = zeros(N);  %  Create a zero matrix of 2*s-by-2*s dimensions
%
%       B_Initial = [Os Is;-Is Os];  % B initial with known structure:
%       %Doesn't do us much good but still structure is known and consistent
%
%end %Initial_B
%%-------------------------------------------------------------------------
%function H = H_Matrix(B_Initial,Og,N)
%% Creates an H skew-symmetric matrix H = [-B -L;L^T 0]
%% Input: B -is initialized B matrix
%% Output: H  - is the H-matrix
%
%         %  Create sub-matrix B
%
%            [Q,R] = qr(randn(N));  % QR factor to extract the orthogonal
%                                   %  matrix Q of dimensions 2*s-by-2*s
%            B = Q'*B_Initial*Q;    % Uniformly scramble B while still
%                                   % perserving structure and E-values
%
%         %  Create a positive definite sub-matrix CC
%
%            CC = rand(N, N) + (N - 1)*eye( N );
```

```
%
%         %  Create L, upper triangle submatrix, from chol(CC)
%
%            L = chol(CC);
%
%         %  Create H = [-B  -L; L^T 0]
%
%            H = [-B  -L; L' 0g];
%
%
%end % H_Matrix
%
%
```

B.2.2 ScaleUniform Test

```
%function [ScaleV, Max_Max20v, MeanMax_20v, StdMax_20v, Max_Min20v, ...
%    MeanMin_20v, StdMin_20v, Max_Max40v, StdMax_40v, MeanMax_40v, ...
%    Max_Min40v, MeanMin_40v, StdMin_40v, Max_Max60v, MeanMax_60v, ...
%    StdMax_60v, Max_Min60v, MeanMin_60v,StdMin_60v, Logical] =
%    ScaleMatrixTest( m, updown)
%
%% Creates data using 15 strictly increasing or decreasing scalar
%% mulitplication of H matrix.  For each up or down scalar, m iterations are
%% of matrices of 20by20, 40by40 and 60by60.  Each iteration then has eig
%% and methodI applied and the evalues are subtracted and then max and min
%% evalues are extracted.  For m iterations, we then find the max-max delta
```

```
%% lambda and max-min delta lambda as well as calculating mean, std
%% deviations for matrix dimensins 20,40 and 60
%
%% Input:  m  -defines how many iterations per matrix dimensions.
%% Input:  updown  - scale 'up' or scale 'down'
%
%% Output: ScaleV  - contains the scale factor for each matrix dimensions
%% Output: Max_Max20v  -Maximum of maximum change in eigenvalue in vector
%                  % form where entries correspond to scale applied to matrix
%                  % with dimensions 20-by-20
%% Output: Max_Min20v -Maximum of minimum change in in eigenvalue in vector
%                  % form where entries correspond to scale applied to matrix
%                  % with dimensions 20-by-20
%% Output: Max_Max40v  - same
%% Output: Max_Min40v  - same
%% Output: Max_Max60v  - same
%% Output: Max_Min60v  - same
%
%% Pseudo code:
%%
%%     I. Create 15 scale up or down using a scalar matrix [D 0;0 I] matrix
%%       such that [D 0;0 I]*[-B -R;R^T 0]*[D 0;0 I]=[D*-B*D D*-R;R^T*D 0]
%%       Store the 15 scalers in vector called ScaleV which will be the first
%%       column of the table
%%
%%       II. For each Scalefactor, it will be applied to H matrix of dimensions
```

```
%%       20 40 and 60.  m number of scaled matrix will be made and max min and
%%       then maxmax and minmin of delta lambda is recorded
%%
%%       Out put will be ScaleV, 20_Max_Max,20_Max_min,40_Max_Max,40_Max_min,
%%       60_Max_Max,60_Max_min, and appropriate stddev and means for each
%
%%
%
%% Created by Wade Rush
%% 13 June 2008   Created
%% 14 June 2008
%% 17 June 2008  incorporating statistics
%
%%Elapsed time for running 15 test with 1  matrix iterations = 97.8394
%%Elapsed time for running 15 test with 10 matrix iterations  971 sec = 16.25 min
%% Tot time: running 15 Scalar test = 30 matrix iters = 3019.5021 for up
%  % scalar:  51 mins
%% Total time to run is 10104 sec which is 2hrs 50 mins for 100 iterations
%%    per scalar level
%
%
%tic
%Logical = updown;
%flag = updown;
%disp('Initiating test run');
%v = 1:7:100;  % Gives us 15 scalefactors when using 7 per step
```

```
%
%   switch (flag)
%       case 'down'
%            ScaleV = (v.^(-2))';  % Scale down factor column vector
%       case 'up'
%            ScaleV = (v.^(2))';  % Scale up factor  column vector
%       otherwise
%           disp('Wrong scale used, choose up or down');
%   end
%
%  NN = length(v);
%  sca = length(ScaleV);
%  count = 0;
%
%    % Initializing table vectors
%    Max_Max20v = zeros(NN, 1);  % Declare vector: stores max_max per scalar
%                                % iteration for 20by20 matrix.
%    Max_Max40v = zeros(NN, 1);  % Same as above
%    Max_Max60v = zeros(NN, 1);  % Same as above
%
%    Max_Min20v = zeros(NN, 1);  % Same as above
%    Max_Min40v = zeros(NN, 1);  % Same as above
%    Max_Min60v = zeros(NN, 1);  % Same as above
%
%    MeanMax_20v = zeros(NN, 1);
%    MeanMax_40v = zeros(NN, 1);
```

```
%    MeanMax_60v = zeros(NN, 1);
%
%    MeanMin_20v = zeros(NN, 1);
%    MeanMin_40v = zeros(NN, 1);
%    MeanMin_60v = zeros(NN, 1);
%
%    StdMax_20v = zeros(NN, 1);
%    StdMax_40v = zeros(NN, 1);
%    StdMax_60v = zeros(NN, 1);
%
%    StdMin_20v = zeros(NN, 1);
%    StdMin_40v = zeros(NN, 1);
%    StdMin_60v = zeros(NN, 1);
%
%    MeanMax_v = zeros(NN,1);  % Collects the mean of the max delta lambda
%                              % for each matrix dim
%    MeanMin_v = zeros(NN,1);  % Collects the mean of the min delta lambda
%                              % for each matrix dim
%    StdMax_v = zeros(NN,1);   % Collects the std of the max delta lambda
%                              %for each matrix dim
%    StdMin_v = zeros(NN,1);   % Collects the std of the min delta lambda
%                              %for each matrix dim
%
%    Max_v = zeros(m,1); % m matrices are created and the max change between
%                        % eig and methodI is recorded m times.
%    Min_v = zeros(m,1); % % m matrices are created and the min change between
```

```
%                       % eig and methodI is recorded m times.
%
%    Max_Max_v = zeros(NN,1); % design to collect then shuffle to Max_Max20v
%                             % Max_Max40v, and Max_Max60v
%
%    Max_Min_v = zeros(NN,1); % design to collect then shuffle to Max_Max20v
%                             % Max_Max40v, and Max_Max60v
%
%
%%
%
%%Tot time: running 15 Scalar test = 10 matrix iters = 990 for down scalar: 17 mins
%%Tot time: running 15 Scalar test = 20 matrix iters = 1980 for down scalar: 33 min
%
%%I.  Design a H matrix with a specific structure
%%II.  Run a for loop  n times " each iteration increses the size of
%%H-matrix". However, there is a limit on matrix H size since eig(vpa(H)) is
%% extremely slow.
%
%       for u=1:3  % Each loop increases the dimensions by 20 with zero the starter.
%
%             if u ==1
%                 s = 20/4;
%             elseif u==2
%                 s = 40/4;
%             else
```

```
%                 s = 60/4;
%             end
%
%         for k=1:NN   % Each k increases or decreases the scale factor applied
%                     % to a generic H matrix with a dimension 20, 40, or 60.
%
%
%
%           for i=1:m  % Number of iterations for each H matrix dimensions
%               %s
%               % i  counter
%               [B_I,Og,N] =Initial_B(s);
%               H = H_Matrix(B_I,Og,N);
%
%               d = ScaleV(k)*ones(N,1); % Uniform scale[d1 0 0;0 d1 0; 0 0 d1]
%               D1=diag(d);  % create a matrix
%               %H;
%               D =[D1 Og; Og D1]; % create matrix same dimensions as H matrix
%               Hscale = D*H*D;  % Scaling H matrix [D1*-B*D1 D1*-R;R^T*D1 0]
%
%               [T, Eigenvalues] = MethodI(Hscale);
%
%                % Determine evalues using Matlab eig(H) QR iteration and
%                %       methodI's iteration
%               [Exact_Evalues, p] = QRSort(Hscale); % List exact evalues:
%                    % QRSort vpa's H  and sorts positive-imaginary evalues
```

```
%                   % in descending order
%
%               % Determine difference between each corresponding E-values.
%
%               diff_lambda = zeros(p,1); % Initialize diff_Lambda vector
%
%               for j=1:p   %  Determine diffence between exact and methodI evalues
%                  diff_lambda(j) = double(abs(Eigenvalues(j)-Exact_Evalues(j)));
%               end
%
%               Max_v(i) = max(diff_lambda);
%               Min_v(i) = min(diff_lambda);
%
%           end
%              count = count + 1;% count is the steps for each scalar iteration
%              Max_Max_v(k)  = max(Max_v);
%              Max_Min_v(k)  = max(Min_v);
%
%              MeanMax_v(k) = mean(Max_v);
%              MeanMin_v(k) = mean(Min_v);
%
%              StdMax_v(k) = std(Max_v);
%              StdMin_v(k) = std(Min_v);
%
%         end
%             count;
```

```
%             count = 0;
%             if u ==1
%                 disp(['length of Max_Maxv for Max_Max20v = '
%                                             num2str(length(Max_Max_v))])
%                 Max_Max20v = Max_Max_v;
%                 Max_Min20v = Max_Min_v;
%                 MeanMax_20v = MeanMax_v;
%                 MeanMin_20v = MeanMin_v;
%                 StdMax_20v = StdMax_v;
%                 StdMin_20v = StdMin_v;
%             elseif u==2
%                 disp(['length of Max_Maxv for Max_Max40v = '
%                                              num2str(length(Max_Max_v))])
%                 Max_Max40v = Max_Max_v;
%                 Max_Min40v = Max_Min_v;
%                 MeanMax_40v = MeanMax_v;
%                 MeanMin_40v = MeanMin_v;
%                 StdMax_40v = StdMax_v;
%                 StdMin_40v = StdMin_v;
%             else
%                 disp(['length of Max_Maxv for Max_Max60v = '
%                                              num2str(length(Max_Max_v))])
%                 Max_Max60v = Max_Max_v;
%                 Max_Min60v = Max_Min_v;
%                 MeanMax_60v = MeanMax_v;
%                 MeanMin_60v = MeanMin_v;
```

```
%                 StdMax_60v = StdMax_v;
%                 StdMin_60v = StdMin_v;
%             end
%       end
%
%%Pseudo Code:
%
%%  I   a.  For each n dimensional H-matrix, we create this matrix m
%%      times by running a for loop.  The size of  m could perhaps be
%%      as large as m = 100"
%
%%      b.  Each H-matrix created, run QRsort() and methodI() and
%%      substract corresponding e-values from each other.
%
%%      c.  Determine the max change and min change in this iteration and
%%      store this into Max_vector and Min_vector which should be size m
%
%%      d.  Sort Max_vector and Min_vector and determining Max-max and
%%      Max-min and store values into Max_Max_v and Max_Min_v which should
%%      be size n
%
%%      e.  Store into Dim_v the size of the matrix for each iteration.
%
%%      f.  Transfer Dim_v, Max_max_v and Max_Min_v into cell array called
%%      UTable where U is uniform matrix distribution.
%
```

```
%%      g.  XLMwrite UTable to excel and then transfer data to Latex.
%
%     disp('Finishing test run');
%
%disp(['Tot time: running ' num2str(NN) ' Scalar test = ' num2str(m) '
%matrix iters = ' num2str(toc) ' for ' updown ' scalar' ]);
%
%end  %Function ScaleMatrixTest
%
%function  [B_Initial, Og,N] = Initial_B(s)
%% Input: s -is the 1/2 the size of B matrix or 1/4 size of H matrix
%% Output:  B_Initial -is the Inital B matrix condition but not final stage
%% Output:  Og - is 2*s-by-2*s matrix which is same size as B , R, R^T or
%           %  0 sub-matrices
%% Output:  N  - is dimensions of B, R, R^T or 0 sub-matrices
%
%     N = 2*s;  %Establishes the dimensions of submatrix B
%
%  % Create B_Initial matrix
%       Is = eye(s);    %  Identity matrix of s-by-s dimensions
%
%       Os = zeros(s);  %  zero Matrix of s-by-s dimensions
%
%       Og = zeros(N);  %  Create a zero matrix of 2*s-by-2*s dimensions
%       % equal to B
%
```

```
%       B_Initial = [Os Is;-Is Os];  % B initial with known structure:
%       %Doesn't do us much good but still structure is known and consistent
%end %Initial_B
%
%function H = H_Matrix(B_Initial,Og,N)
%% Creates an H skew-symmetric matrix H = [-B -R;R^T 0]
%% Input: B -is initialized B matrix
%% Input:  N  - is dimensions of B, R, R^T or 0 sub-matrices
%% Input:  Og - is 2*s-by-2*s matrix which is same size as B , R, R^T or
%           %  0 sub-matrices
%% Output: H  - is the H-matrix
%
%         %  Create sub-matrix B
%
%%-------------------------------------------------------------------------
%%
%%              [Q,R] = qr(randn(N)); % QR factor to extract the orthogonal
%%                                    % matrix Q of dimensions 2*s-by-2*s
%%             B = Q'*B_Initial*Q;    % Uniformly scramble B while still
%                                    % perserving structure and E-values
%%-------------------------------------------------------------------------
%         nn = N;
%            B1 = rand(nn,nn);
%            B = B1 - B1';   % new randomized uniform skew-symmetric B
%         %  Create a positive definite sub-matrix CC
%
```

```
%            CC = rand(N, N) + (N - 1)*eye( N );
%
%         %  Create R, upper triangle submatrix, from chol(CC)
%
%            R = chol(CC);
%
%         %  Create H = [-B  -R; R^T O]
%
%            H = [-B  -R; R' Og];
%
%
%end % H_Matrix
%
%
```

B.2.3 Scale submatrix B Test

```
%function [Scale, Max_Max_20v, MeanMax_20v, StdMax_20v, Max_Min20v, ...
%    MeanMin_20v, StdMin_20v, Max_Max_40v, StdMax_40v, MeanMax_40v, ...
%    Max_Min40v, MeanMin_40v, StdMin_40v, Max_Max_60v, MeanMax_60v, ...
%    StdMax_60v, Max_Min60v, MeanMin_60v,StdMin_60v, Logical1] =
%    ScaleBMatrix2Test(m, updnIncDec)
%
%% Generate a graduated scaled down factor of the submatrix B
%      %where H = [-B -L;L^T 0] then
%% compares eig() and methodI in terms of maxmax  maxmin evalue
%%     differences iterated from
```

```
%% diminsions of 4 to multiples of 4 dim to max-dim in matrix size and
%% we do this m times per matrix dimension.
%
%% Created by Wade Rush
%% 7 July 2008   Not runned yet
%
%% Input:  Max_Dim -describes max dimension is the dimension of the
%%         largest matrix created.  Max_Dim must be a even numbers
%%         which is multiples of 4.
%% Input:  updnIncDec  - scale 'up', 'down', 'increase' or 'decrease'
%
%% Input:  m  -defines how many iterations per matrix dimensions.
%% Output: Dim_v  -contains the dimensions of every new matrix dimension
%%                 in vector form
%% Output: Max_Max_v  -Maximum of maximum change in eigenvalue in vector
%                      % form where entries correspond to size (dimension)
%                      % of matrix
%% Output: Max_Min_v  -Maximum of minimum change in in eigenvalue in vector
%                      % form where entries correspond to size (dimension)
%                      % of matrix
%
%
%%
%% 2 iterations: Run time for one flag or logical1 and only two matrix
%%iteration is 3.3 minutes
%
```

```
%% 100 iterations:  Estimate run time for one logical1 is 2 hrs and
%    45 minutes tic
%
%    Logical1 = updnIncDec;
%
%    disp('Initiating test run for ScaleBMatrix2Test');
%
%    v = 1:7:100;  % Gives us 15 scalefactors when using 7 per step
%
%    NN = length(v);
%    flag = updnIncDec;
%
%%Elapsed time: 7 different Matrix dim,  with each matrix dim iterated
%    %10time is = 30.3599
%
%%I.  Design a H matrix with a specific structure
%%II.  Run a for loop  n times " each iteration increses the size of
%%H-matrix". However, there is a limit on matrix H size since eig(vpa(H))
%% is extremely slow.
%
%               switch (flag)
%                   case 'decrease'% Scale down factor
%                   ScaleV = (v.^(-2))';
%[Scale, Max_Max_20v, MeanMax_20v, StdMax_20v, Max_Min20v, MeanMin_20v,...
%    StdMin_20v,Max_Max_40v, MeanMax_40v, StdMax_40v, Max_Min40v, ...
%    MeanMin_40v,StdMin_40v,Max_Max_60v, MeanMax_60v, StdMax_60v, ...
```

```
%    Max_Min60v, MeanMin_60v,StdMin_60v]=iteration(ScaleV, 'decrease', m);
%
%                   case 'increase'% Scale up factor
%                   ScaleV = (v.^(2))';
%[Scale, Max_Max_20v, MeanMax_20v, StdMax_20v, Max_Min20v, MeanMin_20v,...
%    StdMin_20v,Max_Max_40v, MeanMax_40v, StdMax_40v, Max_Min40v, ...
%    MeanMin_40v,StdMin_40v,Max_Max_60v, MeanMax_60v, StdMax_60v, ...
%    Max_Min60v, MeanMin_60v,StdMin_60v]=iteration(ScaleV, 'increase', m);
%
%                   case 'up'%
%                   ScaleV = 'dummy';
%[Scale, Max_Max_20v, MeanMax_20v, StdMax_20v, Max_Min20v, MeanMin_20v,...
%    StdMin_20v,Max_Max_40v, MeanMax_40v, StdMax_40v, Max_Min40v, ...
%    MeanMin_40v,StdMin_40v,Max_Max_60v, MeanMax_60v, StdMax_60v, ...
%    Max_Min60v, MeanMin_60v,StdMin_60v]=iteration(ScaleV, 'up', m);
%
%
%                   case 'down'%
%                   ScaleV = 'dummy';
%[Scale, Max_Max_20v, MeanMax_20v, StdMax_20v, Max_Min20v, MeanMin_20v,...
%    StdMin_20v,Max_Max_40v, MeanMax_40v, StdMax_40v, Max_Min40v, ...
%    MeanMin_40v,StdMin_40v,Max_Max_60v, MeanMax_60v, StdMax_60v, ...
%    Max_Min60v, MeanMin_60v,StdMin_60v]=iteration(ScaleV, 'down', m);
%
%                 otherwise
%                    disp('Wrong scale used, choose up, down, increase,
```

```
%                                or decrease');
%               end  % switch
%
%%Pseudo Code:
%
%%  I   a.  For each n dimensional H-matrix, we create this matrix
%%      m times by running a "for" loop.  The size of m is generally
%%      set at m = 100 for statistical purposes"
%
%%      b.  Each H-matrix created, run QRsort() and methodI() and substract
%%      corresponding e-values from each other.
%
%%      c.  Determine the max change and min change in this iteration and
%%      store this into Max_vector and Min_vector which should be size m
%
%%      d.  Sort Max_vector and Min_vector and determining Max-max and
%%      Max-min and store values into Max_Max_v and Max_Min_v which should
%%      be size n
%
%%      e.  Store into Dim_v the size of the matrix for each iteration.
%
%%      f.  Transfer Dim_v, Max_max_v and Max_Min_v into cell array called
%%      UTable where U is uniform matrix distribution.
%
%%      g.  XLMwrite UTable to excel and then transfer data to Latex.
%
```

```
%     disp('Finishing test run');
%
%disp(['Elapsed time: ' num2str(NN) ' different Matrix dim,  with each
%     matrix dim iterated 'num2str(m) ' time is = ' num2str(toc) ]);
%disp(['This operations is ' num2str(updnIncDec)]);
%
%end  %Function D_MatrixTest
%
%%-------------------------------------------------------------------------
%
%function  [Max_v1, Min_v1] = HMatrix(D1,H, N)
%% Scales the submatrix B in the matrix H and compares matlab's eig()
%% versus methodI()'s eigenvalues and sets up and extracts the
%%statistical data.
%
%
%               H(1:N,1:N)= D1*H(1:N,1:N)*D1; % scaling sub-Matrix B only
%               Hscale = H;  % Scaled H matrix is renamed Hscale
%               [T, Eigenvalues] = MethodI(Hscale);
%
%                % Determine evalues using Matlab eig(H) QR iteration and
%                %       methodI's iteration
%               [Exact_Evalues, p] = QRSort(Hscale); % List exact evalues:
%                    % QRSort vpa's H  and sorts total number, "p" of
%                    %positive imaginary evalues in descending order.
%
```

```
%               % Determine difference between each corresponding E-values.
%
%               diff_lambda = zeros(p,1); % Initialize diff_Lambda vector
%
%               for j=1:p   %  Determine diffence between exact and methodI
%                  evalues diff_lambda(j) =
%                  double(abs(Eigenvalues(j)-Exact_Evalues(j)));
%               end
%
%               Max_v1 = max(diff_lambda);
%               Min_v1 = min(diff_lambda);
%
%end
%%-------------------------------------------------------------------------
%
%function  [Max_Max20v, MeanMax_20v, StdMax_20v, Max_Min20v, ...
%    MeanMin_20v,StdMin_20v] = Vector20(Max_Max_v, MeanMax_v, ...
%    StdMax_v, Max_Min_v,MeanMin_v, StdMin_v)
%% Function creates statistical data for matrix of dimension 20-by-20
%% which is denoted as dim 20
%
%     Max_Max20v = Max_Max_v;
%     MeanMax_20v = MeanMax_v;
%     StdMax_20v = StdMax_v;
%     Max_Min20v = Max_Min_v;
%     MeanMin_20v = MeanMin_v;
```

```
%     StdMin_20v = StdMin_v;
%end
%%-------------------------------------------------------------------------
%
%function  [Max_Max40v, MeanMax_40v, StdMax_40v, Max_Min40v, ...
%    MeanMin_40v,StdMin_40v] = Vector40(Max_Max_v, MeanMax_v, ...
%    StdMax_v, Max_Min_v,MeanMin_v, StdMin_v)
%% Function creates statistical data for matrix of dimension 40-by-40
%% which is denoted as dim 40
%
%     Max_Max40v = Max_Max_v;
%     MeanMax_40v = MeanMax_v;
%     StdMax_40v = StdMax_v;
%     Max_Min40v = Max_Min_v;
%     MeanMin_40v = MeanMin_v;
%     StdMin_40v = StdMin_v;
%end
%
%%-------------------------------------------------------------------------
%
%function  [Max_Max60v, MeanMax_60v, StdMax_60v, Max_Min60v, ...
%    MeanMin_60v,StdMin_60v] = Vector60(Max_Max_v, MeanMax_v, ...
%    StdMax_v, Max_Min_v,MeanMin_v, StdMin_v)
%% Function creates statistical data for matrix of dimension 60-by-60
%% which is denoted as dim 60
%
```

```
%     Max_Max60v = Max_Max_v;
%     MeanMax_60v = MeanMax_v;
%     StdMax_60v = StdMax_v;
%     Max_Min60v = Max_Min_v;
%     MeanMin_60v = MeanMin_v;
%     StdMin_60v = StdMin_v;
%end
%
%%-------------------------------------------------------------------------
%
%function  [B_Initial,Og,N] = Initial_B(s)
%% Input: s -is the 1/2 the size of B matrix or 1/4 size of H matrix
%% Output:  B_Initial -is the Inital B matrix condition but not final stage
%% Output:  Og - is 2*s-by-2*s matrix which is same size as B , R, R^T or
%           %  0 sub-matrices
%% Output:  N  - is dimensions of B, R, R^T or 0 sub-matrices
%
%     N = 2*s;  %Establishes the dimensions of submatrix B
%
%  % Create B_Initial matrix
%       Is = eye(s);    %  Identity matrix of s-by-s dimensions
%
%       Os = zeros(s);  %  zero Matrix of s-by-s dimensions
%
%       Og = zeros(N);  %  Create a zero matrix of 2*s-by-2*s dimensions
%                       %  which is matrix B dimensions
```

```
%
%       B_Initial = [Os Is;-Is Os];  % B initial with known structure:
%                                    % Doesn't do us much good but still
%                                    % structure is known and consistent
%end %Initial_B
%
%%-----------------------------------------------------------------------
%
%function H = H_Matrix(B_Initial,Og,N)
%
%       nn=N;
%       B1 = rand(N,N);  %uniform distribution N-by-N Matrix
%       B = B1-B1';  % builds a skew-symmetric matrix
%
%       CC = rand(N,N)+(N-1)*eye(N);
%       R = chol(CC);
%       H = [-B -R;R' Og];
%% function H = H_Matrix(B_Initial,Og,N)
%% % Creates an H skew-symmetric matrix H = [-B -R;R^T 0]
%% % Input: B -is randomized B matrix constructed from the initial B =
%% % B_Initial
%% % Input:  N  - is dimensions of B, R, R^T or 0 sub-matrices
%% % Input:  Og - is 2*s-by-2*s matrix which is same size as B , R, R^T
%%            %  or 0 sub-matrices
%% % Output: H  - is the H-matrix
%%
```

```
%%          %  Create sub-matrix B
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%          % Error in this comment out block:  Not really generating
%%          % random B matrix with with random eigenvalues
%% %              random = rand(N);
%% %
%% %              [Q,R] = qr(random); % QR factor to extract the orthogonal
%% %                                  % matrix Q of dimensions 2*s-by-2*s
%% %                                           % changed randn(N) to rand(N)
%% %
%% %              B = Q'*B_Initial*Q; % Uniformly scramble B while still
%% %                                  %  perserving structure and E-values
%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%
%% % Correction block to incorporate true random generator
%% %-------------------------------------------------------------------------
%%         nn= N;
%%
%%  B1 = rand(nn,nn);  %  Generates a random generated 2n-by-2n matrix
%%
%%  B = B1 - B1';
%% %-------------------------------------------------------------------------
%%          %  Create a positive definite sub-matrix CC
%%
%%             CC = rand(N, N) + (N - 1)*eye( N );
%%
```

```
%%          %  Create R, upper triangle submatrix, from chol(CC)
%%
%%             R = chol(CC);
%%
%%          %  Create H = [-B  -R; R^T 0]
%%
%%             H = [-B  -R; R' 0g];
%
%
%end % H_Matrix
%%
%%-------------------------------------------------------------------------
%
%function [Scale, Max_Max_20v, MeanMax_20v, StdMax_20v, Max_Min20v, ...
%    SMeanMin_20v,tdMin_20v,Max_Max_40v, MeanMax_40v, StdMax_40v, ...
%    Max_Min40v, MeanMin_40v,StdMin_40v,Max_Max_60v, MeanMax_60v, ...
%    StdMax_60v, Max_Min60v, MeanMin_60v,StdMin_60v] =
%    iteration(ScaleV, Logical, m)
%
%    %Function iteration() runs 20, 40, and 60 dimensions matrix H, then
%
%    v = 1:7:100;  % Gives us 15 scalefactors when using 7 per step
%    NN = length(v);
%
%
%    Max_Max_v  = zeros(NN,1);  % Initial value vector which will change
```

```
%    LengthMaxMax = length(Max_Max_v)
%    Max_Min_v  = zeros(NN,1);  % Initial value vector which will change
%    MeanMax_v  = zeros(NN,1);  % Collects the mean of the max delta lambda
%                               % for each matrix dim
%    MeanMin_v  = zeros(NN,1);  % Collects the mean of the min delta lambda
%                               % for each matrix dim
%    StdMax_v   = zeros(NN,1);  % Collects the std of the max delta lambda
%                               %for each matrix dim
%    StdMin_v   = zeros(NN,1);  % Collects the std of the min delta lambda
%                               %for each matrix dim
%
%    Max_v = zeros(m,1); % m matrices are created and the max change
%                        % between eig and methodI is recorded m times.
%    Min_v = zeros(m,1); %  m matrices are created and the min change
%                        % between eig and methodI is recorded m times.
%    count = 0;
%
%    if strcmp(Logical, 'up')
%       for ii=2:NN
%           v(ii) = v(ii)/6;
%       end
%    elseif strcmp(Logical, 'down')
%       for ii=2:NN
%           v(ii) = v(ii)/(-6);
%       end
%    else
```

```
%        disp('Using increase or decrease scaleV');
%    end
%    Scale = v;
%
%       for s=5:5:15   % Each s increases H matrix dimension
%                      %  4*s (dim 20, 40, 60)
%        for k=1:NN; % Number of scale factors
%         for i=1:m  % Number of iterations for each H matrix dimensions
%
%                [B_I,Og,N] =Initial_B(s);
%                % N is dimensions of matrix B,
%
%                 H = H_Matrix(B_I,Og,N);
%                  % Randomizer of matrix B is installed here.
%                  % The seed begins here for each iteration
%                    switch (Logical)
%                       case 'increase'
%                       G = ScaleV(k); %  Iterate from 1 to NN and
%                                      % pull out each element
%                       d = G*ones(N,1);% Creates a vector with one
%                                       %scale based on ScaleV(K)
%                       D1=diag(d); % Creates a matrix with diagonal
%                                      % composed of d entries
%                                    % the matrix D1 dimensions equal
%                                     % matrix B's dimensions
%
```

```
%                                        Scale = ScaleV.^2;
%
%                        case 'decrease'
%                        G = ScaleV(k);
%                        d = G*ones(N,1);
%                        D1=diag(d);
%                        Scale = ScaleV.^2;
%
%                        case 'up'
%                        G1 = 1:N;
%                        if k ==1
%                           d = G1.^(k);
%                        else
%                           d = G1.^(k/6);
%                        end
%                        D1 = diag(d);
%                        Scale = 1:15;  %  G(k) is scaled based on k
%
%                        case 'down'
%                        G1 = 1:N;
%                        if k ==1
%                          d = G1.^(-k);
%                        else
%                          d = G1.^(-k/6);
%                        end
%                         D1 = diag(d);
```

```
%                        Scale = 1:15;  %  G(k) is scaled based on k
%                      end %Swith operation ends
%
%                          [Max_v1, Min_v1] = HMatrix(D1,H, N);
%                          Max_v(i) = Max_v1;
%                          Min_v(i) = Min_v1;
%                      end %ends for loop i=1:m  where m usually is 100
%
%                      count = count + 1;  % Counts for each scale
%                                           % factor used
%
%                      MeanMax_v(k) = mean(Max_v);
%                      MeanMin_v(k) = mean(Min_v);
%
%                      StdMax_v(k)  = std(Max_v);
%                      StdMin_v(k)  = std(Min_v);
%
%                      Max_Max_v(k) = max(Max_v);
%                      Max_Min_v(k) = max(Min_v);
%
%                    end
%
%                    if s == 5 % Dimensions 4*s which is 20
%                        disp('dimensions  20');
%                        disp(['length of Max_Maxv for Max_Max20v = '
%                               num2str(length(Max_Max_v))])
```

```
%                           %Max_Max_v;  % display data
%
%     [Max_Max_20v, MeanMax_20v, StdMax_20v, Max_Min20v, MeanMin_20v,...
%     StdMin_20v] = Vector20(Max_Max_v, MeanMax_v, StdMax_v, Max_Min_v,...
%                                       MeanMin_v, StdMin_v);
%
%                           Max_Max_20v   % display data
%
%                       elseif s==10 % Dimensions 4*s which is 40
%                           disp('dimensions  40');
%                           disp(['length of Max_Maxv for Max_Max40v = '
%                                     num2str(length(Max_Max_v))])
%                           %Max_Max_v;   % display data
%
%     [Max_Max_40v, MeanMax_40v, StdMax_40v, Max_Min40v, MeanMin_40v,...
%     StdMin_40v] =  Vector40(Max_Max_v, MeanMax_v, StdMax_v, Max_Min_v,...
%                            MeanMin_v, StdMin_v);
%
%                           Max_Max_40v  % display data
%
%                       elseif s==15 % Dimensions 4*s which is 60
%                           disp('dimensions  60');
%                           disp(['length of Max_Maxv for Max_Max60v = '
%                                    num2str(length(Max_Max_v))])
%                           %Max_Max_v ; % display data
%
```

```
%    [Max_Max_60v, MeanMax_60v, StdMax_60v, Max_Min60v, MeanMin_60v,...
%    StdMin_60v]= Vector60(Max_Max_v, MeanMax_v, StdMax_v, Max_Min_v,...
%                              MeanMin_v, StdMin_v);
%
%                            Max_Max_60v  % display data
%
%                        else
%                         disp('dimensions of matrices test is not...
%                                      of 20, 40 or 60 ');
%
%                        end
%
%                    end
%
%
%end
%
```

B.2.4 Sub-routines to execute these numerical tests

```
%function [Exact_Evalues, t] = QRSort(H)
%% QRSorts determines vpa E_values using matlab's Eig() function and
%% manually sorts the positive imaginary evalues of large vpa floating
%% point accuracy calculated from matlab's eig on symbolic numbers.
%
%% Created by Wade Rush
%%   31 May 08
```

```
%%   1 June 08  Sort function of ig is not quite working so must build a
%%       sort function.  However, symbolic does not recognize
%%       sort probably mainly due to the fact that symbolic
%%       cannot use >, < logic functions.
%
%% Input: Matrix H Skewed symmetric.
%
%% Output:  Exact_Evalues  -Are the positive imaginary eigenvalues in
%                           %descending order of matrix H
%%                          which is a skew-symmetric matrix at vpa
%                           %level of precision.
%% Output:   t   -is the number of positive Exact_evalues in the vector
%                           %array
%        %tic % start the timer
%
%        Hp = vpa(H,32);  %  Increase floating point accuracy
%
%        exact_E_value = eig(Hp);  % eig(): a QR iteration method to
%        %  determine Evalues of  form
%        %  H in symbolic form. Since QR does not have the ability to
%        %  produce pure imaginary E-values
%        %  from a skew-symmetric matrix, due roundoff and truncate
%        %  errors, it cannot sort the
%        %  e-values correctly in descending order.  It also produces
%        %  both plus and minus evalues, we will only use positive
%        %  imaginary evalues
```

```
%
%
%        p = length(exact_E_value);  %
%
%        g = 0;  % Start of a dummy vector
%
%        t = p/2;  % number of positive exact imaginary evalues
%
%        if mod(p,2)~=0
%            disp( ['p is odd which implies t is a fraction, t =
%                        ' num2str(t)] );
%        end
%
%
%
%   % Extracting only the positive exact imag e-values
%        for i = 1:2:p    % eig() appears to not always orders
%                          %e-values from greatest
%            g = [g,exact_E_value(i)];   % to smallest,
%        end
%
%        g = g(:,2:t+1);   % Discards the dummy starter 0 entry.
%
%        Exact_Evalues = imag(g)';  % First cut on ordering e-values and
%                                   % converting E-values to real
%
```

```
%
% % New Sort method applied to symbolic E-values since matlab's sort and
%       %logicals do not  work on symbolic numbers.
%
%        e = length(Exact_Evalues);  % size of symbolic E-value vector
%
%        g = zeros(e,1);  % dummy column vector
%
%        %Create a copy of symbolic E-value vector in the form of a
%        %  double(E-value)  inorder to invoke matlab's sort function.
%
%        for i=1:e
%            g(i)=double(Exact_Evalues(i));
%        end
%
%        g = sort(g);  % create a dummy double e-value vector of ascending
%                      % evalue order
%
%        % Since Sort is only sorts in ascending matter, a for loop is used
%        % to make the order descending.
%
%        E_V = zeros(e,1);
%
%        for i=1:e
%            E_V(i) = g(e+1-i);
%        end
```

```
%        g = E_V;
%
%        % create a vector which describes the permutation which allows the
%        % development of the permutation matrix, P-matrix
%
%        v = 0; % dummy variable
%
%        for i =1:e
%            for j=1:e
%              if double(Exact_Evalues(i)) == g(j);
%                v = [v j];
%              end
%            end
%        end
%        v = v(:,2:e+1);   % Discards the dummy starter 0 entry.
%
%
%        % Build the Pmatrix
%        Pmatrix = zeros(e);
%
%        for j=1:e  % column
%            for i=1:e  % rows
%                if i == v(j)
%                   Pmatrix(i,j) = 1;
%                end
%            end
```

```
%        end
%
%%        Pmatrix
%%         disp('Exact before permutation')
%%         Exact_Evalues
%%         disp('Exact after permutation')
%        Exact_Evalues= Pmatrix*Exact_Evalues;
%
%        %disp(['Elapsed time for diagonalizing a ' num2str(2*t) '
%                    dim matrix = ' num2str(toc)]);
%
%%
%%


%
```

Bibliography

[1] Werner von Braun, Cursader for Space, *Ernst Stuhlinger, Frederick I. Ordway III, Krieger Publishing Company, pg 190-191, 1994*

[2] Kolja Elssel and Heinrich Voss, A modal approach for the Gyroscopic Quadratic Eigenvalue Problem, *European Congress on Computational methods in Applied Sciences and Engineering, pg 2, 24-28 July, 2004*

[3] Beresford N. Parlett, The Symmetric EigenValue Problem, *Society for Industrial and Applied Mathematics, pg 5-6, 1998*

[4] Archara Chaiyakarn, Structure Preserving Algorithms for Computing the Symplectic Singular value Decomposition, PHD Dissertation, *Western Michigan University, pg 16, April 2005*

[5] Lloyd N. Trefethen, Numerical Linear Algebra, *David Bau III, Society for Industrial and Applied Mathematics, pg 187, 1997*

[6] Francoise Tisseur and Karl Meerbergen, The Quadratic Eigenvalue Problem, *Society for Industrial and Applied Mathematics Review, Vol. 43, No. 2 pg 235, 200l*

[7] Sheldon Axler, F.W. Gehring, K.A. Ribet, Linear Algebra Done Right, 2nd Edition, *Mathematics Dept., San Francisco State University, Springer-Verlag, New York, Inc, pg 85, 1997*

[8] http://www.ece.uwaterloo.ca/ ece204/TheBook/04LinearAlgebra/posdef/complete.html; *Univ of Waterloo, Numerical Methods for Electrical and Computer Engineers, Department of Electrical and Computer Engineering, 200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1*

[9] Gene H. Golub, Charles F. Van Loan, Matrix Computation, 3rd Edition, *The Johns Hopkins University Press, Baltimore, Maryland, pg 359 and 455, 1996*

[10] Francoise Tisseur, Backward error and condition of polynomial eigenvalue problems, *Linear Algebra and its Applications, pg 345, 2000*