UNIVERSITY OF MINNESOTA

This is to certify that I have examined this bound copy of a doctoral thesis by

Andrew Allen Anda

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Haesun Park

Name of Faculty Adviser

Signature of Faculty Adviser

Date

GRADUATE SCHOOL

Self-Scaling Fast Plane Rotation Algorithms

A THESIS

SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL

OF THE UNIVERSITY OF MINNESOTA

BY

Andrew Allen Anda

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

February 1995

ABSTRACT

A suite of *self-scaling* fast circular plane rotation algorithms is developed which obviates the monitoring and periodic rescaling necessitated by the standard set of fast plane rotation algorithms. Self-Scaling fast rotations dynamically preserve the normed scaling of the diagonal factor matrix whereas standard fast rotations exhibit divergent scaling. Variations on standard fast rotation matrices are developed and algorithms which implement them are offered. Self-Scaling fast rotations are shown to be essentially as accurate as slow rotations and at least as efficient as standard fast rotations. Computational experimental results utilizing the Cray-2 illustrate the effectively stable scaling exhibited by self-scaling fast rotations. Jacobi-class algorithms with one-sided alterations are developed for the algebraic eigenvalue decomposition using self-scaling fast plane rotations and one-sided modifications. The new algorithms are shown to be both accurate and efficient on both vector and parallel architectures. The utility is described of applying fast plane rotations towards the rank-one update and downdate of least squares factorizations. The equivalence is illuminated of LINPACK, hyperbolic rotation, and fast negatively weighted plane rotation downdating. Algorithms are presented which apply self-scaling fast plane rotations to the QR factorization for stiff least squares problems. Both fast and standard Givens rotation-based algorithms are shown to produce accurate results regardless of row sorting even with extremely heavily row weighted matrices. Such matrices emanate, e.g., from equality constrained least squares problems solved via the weighting

method. The necessity of column sorting is emphasized. Numerical tests expose the Householder QR factorization algorithm to be sensitive to row sorting and it yields less accurate results for greater weights. Additionally, the modified Gram-Schmidt algorithm is shown to be sensitive to row sorting to a notably significant but lesser degree. Self-Scaling fast plane rotation algorithms, having competitive computational complexities, must therefore be the method of choice for the QR factorization of stiff matrices. Timing results on the Cray 2, [XY]M/P, and C90, of rotations both in and out of a matrix factorization context are presented. Architectural features that can best exploit the advantageous features of the new fast rotations are subsequently discussed.

ACKNOWLEDGEMENTS

This thesis is directly and indirectly the result of the the influences of many fine mentors, family members, friends, and institutions–too many to list comprehensively. Foremost amongst my mentors is my advisor, Professor Haesun Park. To Haesun I offer my unbounded gratitude and appreciation for her guidance, support, patience, enthusiasm, pedagogy, insight, encouragement, and friendship. I cannot imagine anyone better serving me as an advisor and mentor.

I thank the members of my Ph.D. committee: professors Ahmed Sameh and David Fox of my department of computer science, and professors E. B. Lee and Keshab Parhi of the department of electrical engineering.

I attended this school on the strength of this department in numerical computation, and I was certainly not disappointed. I was fortunate to be enriched by the teaching of Professors Park, Fox, Boley, and Chronopolous in this department, and Professor Gulliver in Mathematics. In an administrative capacity, I thank Professors Munro, Stein, Fox, and Sameh for their support.

I have had numerous enriching and encouraging interactions with members of the field of numerical linear algebra in person and through correspondence. I look forward with pleasure to continuing my interaction with this community.

I thank Drs. Yiming Yang and Christopher Chute, M.D., and Geoff Atkin of the Mayo Foundation. I also thank the Army High Performance Computing Research Center at the University of Minnesota for its support.

Three prominent mentors I had the pleasure of working for in the interim between my undergraduate and graduate education were Drs. James Valentini and James (Mac) Hyman at the Los Alamos National Laboratory and Dr. Jack Dongarra at the Argonne National Laboratory. The experience, insight, and guidance I received while working with these three mentors was invaluable and instrumental in my subsequent graduate focus and success. They have my gratitude and appreciation.

I had numerous fine professors at my Northeastern Illinois University, where I earned my undergraduate degrees. Most notably I thank Professor Cibulskis of the mathematics department who illuminated the field of numerical computation for me, forever sparking my interest; and Professor Charles Nissim-Sabat of the physics department who guided my senior project and was too a fine friend. And I thank Steve Wampler at the University of Arizona who enthusiastically sparked my interest in computing which led to my subsequent change of majors and universities.

Before college I had enriching experiences At the Adler Planetarium in classes taught by Dr. Carlson and also the Astro Science Workshop of Dr. J. Allen Hynek. I would also like to express my thanks to Frank C. Olson most notably and the other teachers at Lane Tech H.S. who volunteered their efforts assisting in my participation in the Science and Math Conference and Symposium through their guidance and encouragement.

Finally, I'd like to offer my warmest thanks and express my indebtedness to my parents, grandparents, and my other family members, and to my friends for years of unswerving support and encouragement. It is to my family and to my friends that I dedicate this thesis.

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

TABLE OF CONTENTS

Chapter I

INTRODUCTION

1.1 Plane Rotations

A plane rotation $J(\theta, p, q)$ of order n through an angle θ in a (p, q) plane is the same as the identity matrix I_n, except for the four elements at the intersections of the p-th and q-th rows and columns. The general form of an $n \times n$ rotation matrix $J(\theta, p, q)$ in the (p, q) plane, which we will denote as J, is

$$J = J(\theta, p, q) = \begin{bmatrix} I_{p-1} & 0 & 0 & 0 & 0 \\ 0 & c & 0 & -s & 0 \\ 0 & 0 & I_{q-p-1} & 0 & 0 \\ 0 & s & 0 & c & 0 \\ 0 & 0 & 0 & 0 & I_{n-q} \end{bmatrix}, \tag{1.1}$$

where $c \equiv \cos\theta, s \equiv \sin\theta$. The four elements must have an odd number of positive and negative elements, in any ordering, for the matrix to represent an orthogonal transformation. If the signs on the diagonal match, then the transformation represents a rotation, otherwise it is a reflection. Plane rotations are used in various algorithmic contexts, e.g., the QR factorization, the Jacobi and QR algorithms for the eigendecompositions, the Hestenes algorithm for the singular value decomposition, and reduction to Hessenberg form. Although these algorithms differ in the

purpose and range of the angles, we can classify them largely into two cases : those which apply similarity transformations

$$\tilde{X} = J(\theta, p, q)^T X J(\theta, p, q) \tag{1.2}$$

and others which apply one-sided transformations

$$\tilde{X} = XJ(\theta, p, q) \text{ or } \tilde{X} = J^T(\theta, p, q)X. \tag{1.3}$$

We will use the notations X_{pq} and J_{pq} to denote the 2×2 submatrices of X and $J(\theta, p, q)$ in the (p, q) plane, respectively, i.e.,

$$X_{pq} = \begin{bmatrix} x_{pp} & x_{pq} \\ x_{qp} & x_{qq} \end{bmatrix} \text{ and } J_{pq} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix}.$$

Fast rotations were developed with the motivations to reduce the number of scalar-vector multiplications and eliminate square roots from the calculation of the plane rotations. Gentleman [Gen73b] first formulated a method for a fast Givens rotation. Hammarling [Ham74] later modified that formulation into several computational schemes and pointed out that the method can be applied to similarity transformations. There are other types of fast rotations which are developed for the purpose of reducing the number of multiplications in a product of plane rotations [Bar85] or to reduce the device area on a systolic array [GS91]. However, a literature survey of research and production algorithms which utilize plane rotations has shown a pronounced avoidance of the fast plane rotation. This is partly a consequence of the fact that careful monitoring for the prevention of underflow and overflow is not conveniently achieved wherever the standard fast plane rotation algorithm is utilized [BI87, GVL89]. Hammarling [Ham74] suggested that underflow can be avoided either

by storing the exponent separately, by normalizing occasionally, or by performing row (or column) interchanges. Separate storage of the exponent is clearly not efficient for currently popular high level languages. Row (or column) interchanges introduce an overhead. Moreover, occasional normalization can be problematic to implement since a nontrivial amount of monitoring is necessary in order to successfully implement fast Givens transformations.

Chapter II

FAST ROTATIONS

2.1 Established Fast Rotations

Suppose one step of a one-sided transformation via a rotation $J(\theta, p, q)$ gives

$$\tilde{X} = XJ(\theta, p, q) \tag{2.1}$$

as in the one-sided Jacobi [EP90, PE90] and the Hestenes[Hes58] algorithms, and in the QR factorization (transposed). The essential idea of a fast rotation is that the number of multiplications is reduced by keeping the matrix X in the factored form YD, where D is a diagonal matrix and Y is accordingly scaled, and these two factors are updated separately. The calculation of the product of the two factors may be postponed until the explicit result is required. The diagonal matrix D is initialized as the identity matrix. The secondary advantage of the fast rotation is that the square roots for the computation of cosine and sine can be eliminated [Rat82]. If $X = YD$, then the rotation from the right by J can be represented as

$$\tilde{X} = XJ = YDJ = YF(p, q)\tilde{D} = \tilde{Y}\tilde{D} \tag{2.2}$$

where $\tilde{D}$ is a diagonal matrix and $F(p,q)$ is defined as

$$F(p,q) \equiv \begin{bmatrix} I_{p-1} & 0 & 0 & 0 & 0 \\ 0 & f_{pp} & 0 & f_{pq} & 0 \\ 0 & 0 & I_{q-p-1} & 0 & 0 \\ 0 & f_{qp} & 0 & f_{qq} & 0 \\ 0 & 0 & 0 & 0 & I_{n-q} \end{bmatrix} \tag{2.3}$$

We will call $F(p,q)$ a *fast rotation* if the choices of f_{pp}, f_{pq}, f_{qp}, and f_{qq} result in halving the number of multiplications in applying the rotation J. If a typical step is represented as

$$X^{(k+1)} = X^{(k)} J^{(k)}, \tag{2.4}$$

then

$$X^{(k+1)} = Y^{(k+1)} D^{(k+1)} = Y^{(1)} F^{(1)} \ldots F^{(k)} D^{(k+1)} = X^{(1)} J^{(1)} \ldots J^{(k)} \tag{2.5}$$

and, we have

$$J^{(i)} = D^{(i)^{-1}} F^{(i)} D^{(i+1)}. \tag{2.6}$$

In the case of the two-sided transformations, if $X = DYD$, then

$$\tilde{X} = J^T X J = J^T D Y D J = \tilde{D} F^T Y F \tilde{D} = \tilde{D} \tilde{Y} \tilde{D}. \tag{2.7}$$

If a typical step is represented as $X^{(k+1)} = J^{(k)^T} X^{(k)} J^{(k)}$, then

$$\begin{aligned} X^{(k+1)} &= D^{(k+1)} Y^{(k+1)} D^{(k+1)} \\ &= D^{(k+1)} F^{(k)^T} \ldots F^{(1)^T} Y^{(1)} F^{(1)} \ldots F^{(k)} D^{(k+1)} = J^{(k)^T} \ldots J^{(1)^T} X^{(1)} J^{(1)} \ldots J^{(k)} \end{aligned}$$

and, again, we have

$$J^{(i)} = {D^{(i)}}^{-1} F^{(i)} D^{(i+1)}. \tag{2.8}$$

We will discuss only column oriented one-sided transformations since we have implemented the new algorithms in Fortran in which the column oriented one-sided transformations give higher efficiency and also our work has been motivated by one-sided Jacobi algorithms [PE90]. The results for row oriented and the two-sided transformations follow easily.

2.1.1 Standard Fast Rotations

Suppose a fast rotation transforms $X = YD$ into $\tilde{X} = \tilde{Y}\tilde{D}$. The transformation can be shown as follows:

$$\begin{aligned}
[\tilde{x}_p, \tilde{x}_q] &= [x_p, x_q] \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \\
&= [y_p, y_q] \begin{bmatrix} d_p & 0 \\ 0 & d_q \end{bmatrix} \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \\
&= [y_p, y_q] \begin{bmatrix} f_{pp} & f_{pq} \\ f_{qp} & f_{qq} \end{bmatrix} \begin{bmatrix} \tilde{d}_p & 0 \\ 0 & \tilde{d}_q \end{bmatrix} \\
&= [\tilde{y}_p, \tilde{y}_q] \begin{bmatrix} \tilde{d}_p & 0 \\ 0 & \tilde{d}_q \end{bmatrix},
\end{aligned}$$

where x_i denotes the ith column of X.

There are several ways to choose $F_{pq} = \begin{bmatrix} f_{pp} & f_{pq} \\ f_{qp} & f_{qq} \end{bmatrix}$ and $\tilde{D}$ so that the number of multiplications is reduced by half compared to the standard rotation. The two most

commonly used F_{pq} are either of the type $\begin{bmatrix} 1 & \alpha \\ \beta & 1 \end{bmatrix}$ or $\begin{bmatrix} \beta & 1 \\ 1 & \alpha \end{bmatrix}$. With the first type, we have

$$[\tilde{x}_p, \tilde{x}_q] = [y_p, y_q] \begin{bmatrix} 1 & -\alpha \\ \beta & 1 \end{bmatrix} \begin{bmatrix} cd_p & 0 \\ 0 & cd_q \end{bmatrix},$$

thus

$$\left\{ \begin{array}{lcl} \tilde{d}_p & \Leftarrow & cd_p \\ \tilde{d}_q & \Leftarrow & cd_q \\ & & \\ \tilde{y}_p & \Leftarrow & y_p + \beta y_q \\ \tilde{y}_q & \Leftarrow & y_q - \alpha y_p \end{array} \right\} \tag{2.9}$$

where $\alpha = t\frac{d_p}{d_q}$, $\beta = t\frac{d_q}{d_p}$ and $t = \tan\theta$.

When $|\theta| > \frac{\pi}{4}$, especially when $|c| \ll 1$, successive multiplications by small factors in D can quickly lead to underflow. To bound the maximum decrease in the diagonal factor matrix D, one must use an alternative formulation of the fast rotation which updates the diagonal elements of D with sines rather than cosines:

$$[\tilde{x}_p, \tilde{x}_q] = [y_p, y_q] \begin{bmatrix} \beta & -1 \\ 1 & \alpha \end{bmatrix} \begin{bmatrix} sd_q & 0 \\ 0 & sd_p \end{bmatrix},$$

thus

$$\left\{ \begin{array}{lcl} \tilde{d}_p & \Leftarrow & sd_q \\ \tilde{d}_q & \Leftarrow & sd_p \\ & & \\ \tilde{y}_p & \Leftarrow & y_q + \beta y_p \\ \tilde{y}_q & \Leftarrow & -y_p + \alpha y_q \end{array} \right\} \tag{2.10}$$

where $\alpha = \frac{1}{t}\frac{d_q}{d_p}$ and $\beta = \frac{1}{t}\frac{d_p}{d_q}$.

Although, for any rotation, the decrease in magnitude of each element of the diagonal factor D can be bounded by $1/\sqrt{2}$, with the use of the above two formulations, the diagonal elements of D are reduced each time and may eventually cause underflow.

2.1.2 Modified Fast Rotations

De Rijk [dR89] further developed the fast rotation for the Hestenes algorithm for computing the singular value decomposition on vector processors to eliminate a temporary copy of one of the columns. The idea is that the expression $\tilde{y}_p \Leftarrow y_p + \beta y_q$ in (2.9) can be rearranged as $y_p = \tilde{y}_p - \beta y_q$, thus, substituting it into the other triad, we get

$$\begin{aligned}
\tilde{y}_q &= y_q - \alpha y_p \\
&= y_q - \alpha(\tilde{y}_p - \beta y_q) \\
&= \left(y_q - \left(\frac{\alpha}{1+t^2}\right)\tilde{y}_p\right)\left(1+t^2\right) \\
&= \left(y_q - (\alpha c^2)\tilde{y}_p\right)(c^{-2})
\end{aligned}$$

Letting the diagonal element d_q of D take care of the value c^{-2}, we get $\tilde{d}_q \Leftarrow \tilde{d}_q/c^2 = d_q/c$ and

$$[\tilde{x}_p, \tilde{x}_q] = [y_p, y_q] \begin{bmatrix} 1 & 0 \\ \beta & 1 \end{bmatrix} \begin{bmatrix} 1 & -\alpha \\ 0 & 1 \end{bmatrix} \begin{bmatrix} cd_p & 0 \\ 0 & c^{-1}d_q \end{bmatrix}$$

$$
(2.11) \qquad \left\{ \begin{array}{rcl} \tilde{d}_p & \Leftarrow & c d_p \\ \tilde{d}_q & \Leftarrow & c^{-1} d_q \\ & & \\ \tilde{y}_p & \Leftarrow & y_p + \beta y_q \\ \tilde{y}_q & \Leftarrow & y_q - \alpha \tilde{y}_p \end{array} \right\},
$$

where $\alpha \Leftarrow \alpha c^2 = cs\frac{d_p}{d_q}$, and $\beta = t\frac{d_q}{d_p}$. In the Hestenes algorithm for computing the singular value decomposition, the angle can be always chosen in $[-\pi/4, \pi/4]$, thus $1/\sqrt{2} \leq |c|$.

The same procedures which yield expression (2.11) also yield a modification of expression (2.10) for angles $\theta \in [\pi/4, \pi/2]$ in magnitude to scale the diagonal factor:

$$
[\tilde{x}_p, \tilde{x}_q] = [y_p, y_q] \begin{bmatrix} \beta & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & \alpha \\ 0 & -1 \end{bmatrix} \begin{bmatrix} s d_q & 0 \\ 0 & s^{-1} d_p \end{bmatrix}
$$

$$
(2.12) \qquad \left\{ \begin{array}{rcl} \tilde{d}_p & \Leftarrow & s d_q \\ \tilde{d}_q & \Leftarrow & s^{-1} d_p \\ & & \\ \tilde{y}_p & \Leftarrow & y_q + \beta y_p \\ \tilde{y}_q & \Leftarrow & -y_p + \alpha \tilde{y}_p \end{array} \right\},
$$

where $\alpha = cs\frac{d_q}{d_p}$, and $\beta = \frac{1}{t}\frac{d_p}{d_q}$.

For the fast rotations, presented in (2.9) and (2.10), the order in which the columns are updated is inconsequential providing that the correct vector has been copied for reuse. However, de Rijk's modified fast rotation presented in Eqn. (2.11) requires that the p-th column be updated prior to the updating of the q-th column. A variant of de Rijk's rotation follows naturally from interchanging the order of the

column updates, for angles $|\theta| \leq \pi/4$:

$$[\tilde{x}_p, \tilde{x}_q] = [y_p, y_q] \begin{bmatrix} 1 & -\alpha \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \beta & 1 \end{bmatrix} \begin{bmatrix} c^{-1}d_p & 0 \\ 0 & cd_q \end{bmatrix}$$

$$\left\{ \begin{array}{lcl} \tilde{d}_q & \Leftarrow & cd_q \\ \tilde{d}_p & \Leftarrow & c^{-1}d_p \\ & & \\ \tilde{y}_q & \Leftarrow & y_q - \alpha y_p \\ \tilde{y}_p & \Leftarrow & y_p + \beta \tilde{y}_q \end{array} \right\} \tag{2.13}$$

where $\alpha = t\frac{d_p}{d_q}$, and $\beta = cs\frac{d_q}{d_p}$, and, for angles $\theta \in [\pi/4, \pi/2]$ in magnitude,

$$[\tilde{x}_p, \tilde{x}_q] = [y_p, y_q] \begin{bmatrix} 0 & -1 \\ 1 & \alpha \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \beta & 1 \end{bmatrix} \begin{bmatrix} s^{-1}d_q & 0 \\ 0 & sd_p \end{bmatrix}$$

$$\left\{ \begin{array}{lcl} \tilde{d}_q & \Leftarrow & sd_p \\ \tilde{d}_p & \Leftarrow & s^{-1}d_q \\ & & \\ \tilde{y}_q & \Leftarrow & -y_p + \alpha y_q \\ \tilde{y}_p & \Leftarrow & y_q + \beta \tilde{y}_q \end{array} \right\}, \tag{2.14}$$

where $\alpha = \frac{1}{t}\frac{d_q}{d_p}$, and $\beta = cs\frac{d_p}{d_q}$.

We call the new fast rotations presented in (2.11)–(2.14) as the chained fast rotations. The following simplified representations of three different methods for applying a plane rotation, $\begin{bmatrix} c & -s \\ s & c \end{bmatrix}$, to update the two vectors, w and $v \in \mathcal{R}^n$ illustrate that the new fast rotations can be chained:

Standard rotation:

$$[\tilde{v}, \tilde{w}] = [v, w] \begin{bmatrix} c & -s \\ s & c \end{bmatrix} = [cv + sw, -sv + cw] \tag{2.15}$$

Standard fast rotation:

$$[\tilde{v}, \tilde{w}] = [v, w] \begin{bmatrix} 1 & \alpha \\ \beta & 1 \end{bmatrix} = [v + \beta w, w + \alpha v] \tag{2.16}$$

Chained fast rotation:

$$[\tilde{v}, \tilde{w}] = [v, w] \begin{bmatrix} 1 & 0 \\ \beta & 1 \end{bmatrix} \begin{bmatrix} 1 & \alpha \\ 0 & 1 \end{bmatrix} = [v + \beta w, w + \alpha(v + \beta w)] \tag{2.17}$$

The computational efficiency of the chained fast plane rotation over the standard fast plane rotation can result from the elimination of the temporary vector store and read which are necessary for the standard fast plane rotation.

2.1.2.1 Slow Chained Rotations

The slow rotation can also be chained to eliminate a vector temporary. If

$$[\tilde{x}_p, \tilde{x}_q] = [x_p, x_q] \begin{bmatrix} c & -s \\ s & c \end{bmatrix},$$

then

$$\begin{aligned} \tilde{x}_p &= cx_p + sx_q \\ \tilde{x}_q &= cx_q - sx_p \\ &= (c + t)x_q - t\tilde{x}_p \qquad if \quad |\theta| < \pi/4 \\ &= t^{-1}\tilde{x}_p - (ct^{-1} + s)x_p \quad if \quad |\theta| > \pi/4. \end{aligned} \tag{2.18}$$

The chained slow rotation still requires the same number of multiplication steps as the non-chained slow rotation except for the special case of $\pi/4$ at which point it requires only three. As with the fast rotation, care must be taken to always implement the chained slow rotation with respect to the appropriate angle formulation.

Further reformulations of the plane rotation are listed in Appendix A.

2.2 Self-Scaling Algorithms

Using the preceding expressions for fast plane rotations, we now construct algorithms which dynamically scale the elements of the diagonal matrix to be close to one. In the standard fast plane rotation, presented in (2.9)–(2.10), both diagonal elements are diminished at each rotation. In the chained fast rotations, presented in (2.11)–(2.14), one diagonal element is augmented by the same factor as the other diagonal element is diminished. Expressions (2.11) and (2.12) diminish the diagonal element having the smaller index and augment the element having the larger index. This way, the smaller the index is, the more often the corresponding diagonal element will be diminished. Likewise, the larger the index is, the more often the corresponding diagonal element will be augmented. The resulting distribution, after one or more cycles, tends to have the largest diagonal entries in the highest indexed locations, the smallest diagonal entries in the lowest indexed locations. We develop two fast plane rotation algorithms that avoid this unbalanced diagonal element distribution by incorporating dynamic scaling: a four way branching algorithm, and a two way branching algorithm.

The motivation for the four way branch algorithm is to force each diagonal element to be close to unity. Thus, the choice of the fast rotation expression will be based on whether each of the two diagonal elements is either greater than or less than unity in magnitude. This allows four possibilities which yield an absolute bound $1/\sqrt{2} \leq |d_i| \leq \sqrt{2}$ on the magnitude of the diagonal elements d_i's after any rotation. Table 2.1 presents the essential idea in the four way branch algorithms.

The motivation for the two way branch algorithm is to simplify the four way

Table 2.1: **Four Way Branch**

expression		$\lvert d_p\rvert \geq 1$	$\lvert d_p\rvert < 1$
$\lvert d_q\rvert \geq 1$	$\lvert\theta\rvert \leq \frac{\pi}{4}$	$\tilde{d}_p \Leftarrow cd_p$ $\tilde{d}_q \Leftarrow cd_q$	$\tilde{d}_p \Leftarrow d_p/c$ $\tilde{d}_q \Leftarrow cd_q$
	$\lvert\theta\rvert > \frac{\pi}{4}$	$\tilde{d}_p \Leftarrow sd_q$ $\tilde{d}_q \Leftarrow sd_p$	$\tilde{d}_p \Leftarrow sd_q$ $\tilde{d}_q \Leftarrow d_p/s$
$\lvert d_q\rvert < 1$	$\lvert\theta\rvert \leq \frac{\pi}{4}$	$\tilde{d}_p \Leftarrow cd_p$ $\tilde{d}_q \Leftarrow d_q/c$	$\tilde{d}_p \Leftarrow 1$ $\tilde{d}_q \Leftarrow 1$
	$\lvert\theta\rvert > \frac{\pi}{4}$	$\tilde{d}_p \Leftarrow d_q/s$ $\tilde{d}_q \Leftarrow sd_p$	$\tilde{d}_p \Leftarrow 1$ $\tilde{d}_q \Leftarrow 1$

branch algorithm and avoid the slow rotation, while constraining the diagonal elements from deviating far from unity. The relative sizes of the two diagonal elements, d_p and d_q, are compared before each rotation in a (p, q) plane is applied, and an appropriate choice from equations (2.11) – (2.14) is applied to achieve tighter clustering of the diagonal elements about unity. This idea is summarized in Table 2.2, where the typical fast rotation F_{pq} for each case is also presented. In the two way branch algorithm, if two diagonal elements are both greater than unity, the smaller will be augmented. Likewise, the larger of two sub-unity elements will be diminished. Thus, it allows larger deviations in the diagonal elements compared to the four way branch algorithms. However, the results of numerical tests show that the bound in the two way branch algorithms is kept within a small range around 1.

Table 2.2: **Two Way Branch**

expression	$\lvert\theta\rvert \le \frac{\pi}{4}$	$\lvert\theta\rvert > \frac{\pi}{4}$
$\lvert d_p\rvert \ge \lvert d_q\rvert$	$\tilde{d}_p \Leftarrow cd_p$ $\tilde{d}_q \Leftarrow d_q/c$ $\alpha = csd_p/d_q$ $\beta = td_q/d_p$ $F_{pq} = \begin{pmatrix} 1 & 0 \\ \beta & 1 \end{pmatrix}\begin{pmatrix} 1 & -\alpha \\ 0 & 1 \end{pmatrix}$	$\tilde{d}_p \Leftarrow d_q/s$ $\tilde{d}_q \Leftarrow sd_p$ $\alpha = d_q/(td_p)$ $\beta = csd_p/d_q$ $F_{pq} = \begin{pmatrix} 0 & -1 \\ 1 & \alpha \end{pmatrix}\begin{pmatrix} 1 & 0 \\ \beta & 1 \end{pmatrix}$
$\lvert d_p\rvert < \lvert d_q\rvert$	$\tilde{d}_p \Leftarrow d_p/c$ $\tilde{d}_q \Leftarrow cd_q$ $\alpha = td_p/d_q$ $\beta = csd_q/d_p$ $F_{pq} = \begin{pmatrix} 1 & -\alpha \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ \beta & 1 \end{pmatrix}$	$\tilde{d}_p \Leftarrow sd_q$ $\tilde{d}_q \Leftarrow d_p/s$ $\alpha = csd_q/d_p$ $\beta = d_p/(td_q)$ $F_{pq} = \begin{pmatrix} \beta & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} 1 & \alpha \\ 0 & -1 \end{pmatrix}$

The new fast rotations presented in relations (2.11) – (2.14) and employed in the two way branch algorithms have the typical form,

$$F_{pq} = \begin{bmatrix} 1 & \alpha \\ \beta & 1+\alpha\beta \end{bmatrix},$$

or F_{pq} with its rows and/or columns permuted. Although only one element of F_{pq} is unity, the total multiplications required for the transformation by F_{pq} is the same as

those for the standard fast rotations. This is because F_{pq} can be applied in two steps as shown in (2.11) – (2.14), e.g.,

$$F_{pq} = \begin{bmatrix} 1 & \alpha \\ \beta & 1+\alpha\beta \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \beta & 1 \end{bmatrix} \begin{bmatrix} 1 & \alpha \\ 0 & 1 \end{bmatrix}.$$

The implementation results are obtained for the Jacobi algorithm for symmetric eigenvalue decomposition, the Hestenes algorithm for the singular value decomposition, and the QR factorization. The purpose of the tests is to compare the deviations from unity of the elements in the diagonal factor matrix in the standard fast rotation, de Rijk's fast rotation, and the new fast rotation presented in this section. For the eigenvalue and singular value decompositions, the total number of sweeps was assigned to be $\log n$, where n is the matrix order. The test matrices were generated with random numbers in the interval $[-1, 1]$ having a uniform distribution. In Tables 2.3, 2.4, and 2.5, the entries are the average, for 32 tests, of the minimum and maximum of the diagonal elements throughout each test with $\log_{10}$ scaling. The standard deviations, also with $\log_{10}$ scaling, are within the parentheses. For the standard fast rotation, only the minimum is shown since the diagonal elements, which are initially 1, are only decreasing. Note that for all the tests (for the matrices of order up to 128 for the symmetric eigenvalue decomposition and the singular value decomposition, and for the matrices of order up to 256 for the QR factorization), the averages for the extremes of the elements in the diagonal factor of the new algorithms stay in $[1/3, 3]$. Also, the standard deviations for the new algorithms are extremely small. The average of the minimum diagonal elements in the standard fast algorithms diminished to as small as $10^{-4.62}$ and the range of the average extremes of the diagonal elements in the diagonal factor matrix in the de Rijk's method was as large as $[10^{-3.70}, 10^{3.73}]$.

We expect that as the matrix order becomes larger, the advantage of the new fast rotations will become more pronounced. The computational experiments indicate that only the new chained fast rotation algorithm can control the sizes of elements in the diagonal factor for larger matrices.

Table 2.3: **Symmetric Eigenvalue Decomposition** (32 tests)

$\log_{10}$(scaling)	$N = 32$	$N = 64$	$N = 128$
min(standard)	-1.768 (0.2026)	-2.832 (0.2487)	-4.282 (0.3253)
min(de Rijk)	-1.396 (0.1404)	-2.237 (0.2538)	-3.539 (0.4733)
max(de Rijk)	1.344 (0.2209)	2.157 (0.2504)	3.369 (0.3795)
min(new)	-0.2024 (2.271E-2)	-0.2175 (2.490E-2)	-0.2330 (1.796E-2)
max(new)	0.2014 (1.135E-2)	0.2183 (2.088E-2)	0.2332 (2.189E-2)

Table 2.4: **Singular Value Decomposition** (32 tests)

$\log_{10}$(scaling)	$N = 32$	$N = 64$	$N = 128$
min(standard)	-1.884 (0.1984)	-3.052 (0.2149)	-4.620 (0.3295)
min(de Rijk)	-1.567 (0.2098)	-2.515 (0.2828)	-3.703 (0.3496)
max(de Rijk)	1.410 (0.2136)	2.455 (0.3099)	3.733 (0.3759)
min(new)	-0.2133 (2.259E-2)	-0.2260 (1.904E-2)	-0.2384 (1.497E-2)
max(new)	0.2066 (2.739E-2)	0.2198 (2.272E-2)	0.2308 (1.585E-2)

Table 2.5: QR **Factorization** (32 tests)

$\log_{10}$(scaling)	$N = 64$	$N = 128$	$N = 256$
min(standard)	-2.692 (0.4532)	-3.152 (0.3360)	-4.258 (0.5092)
min(de Rijk)	-0.8387 (8.995E-2)	-0.9976 (8.916E-2)	-1.141 (7.915E-2)
max(de Rijk)	1.241 (0.1949)	1.511 (0.1736)	1.858 (0.2492)
min(new)	-0.3727 (7.762E-2)	-0.3943 (7.234E-2)	-0.4491 (7.085E-2)
max(new)	0.3614 (7.7540E-2)	0.3887 (8.318E-2)	0.4404 (6.147E-2)

Each entry in Tables 2.3 - 2.5 denotes $\log_{10}$ *scaling of the minimum or maximum diagonal elements in the diagonal factor matrix throughout each test.*
The standard deviations, within the parentheses, are also scaled to $\log_{10}$.

Legend:	
standard	The standard fast rotation.
de Rijk	The de Rijk fast rotation.
new	The new chained fast rotation with dynamic scaling.

2.2.0.2 A Scaling *Drift* Heuristic

The original de Rijk rotation, expression (2.11) and expression (2.12), as we established previously, diminishes the diagonal element having the lesser index and augments the index having the larger index. Although, there is no net scaling when the effect of the rotation is averaged between the two diagonal elements, the smaller the index, the more often that index's corresponding diagonal element will be diminished rather than augmented. And correspondingly, the larger the index, the more often that index's corresponding diagonal element will be augmented. The resulting distribution, after one or more cycles, tends to have the largest diagonal entries in the highest indexed locations, the smallest diagonal entries in the lowest indexed locations, and a rather chaotic non-monotonic distribution of magnitudes in the middle section of the diagonal. One may conceptualize this process as having a scaling *drift* which is roughly a function of index and cycle count.

The fundamental conceptual motivation for the two way branch algorithm is to counteract a scaling *drift* in one index direction with a *counter-drift* in the opposing direction. To this end, we ran numerical experiments, varying only in the algorithm by which we chose to *drift* or *counter-drift*. The five algorithms are:

1. No *counter-drift* (the experimental control).

2. Toggle between *drift* and *counter-drift* at every new every cycle.

3. Toggle between *drift* and *counter-drift* at every new every sweep.

4. Toggle between *drift* and *counter-drift* randomly.

5. Toggle between *drift* and *counter-drift* such that if $d_p > d_q$, *drift*, else, *counter-drift*.

The de Rijk rotation, expression (2.11) and expression (2.12), as we saw in the previous section, tends to have smaller scaled diagonal factor elements corresponding to smaller column indices; and larger scaled diagonal factor elements corresponding to larger column indices. The motivation behind the "Counter-drift" rotation is to counteract the index correlated skew of scale magnitudes. In this counter-drift algorithm, the index correlated skew is in the opposite direction,i.e., the scaling magnitude of the lower indexed diagonal factor elements will instead increase, and that of the higher indexed elements will decrease. This rotation is essentially a mirrored image of the de Rijk rotation, expression (2.11):

The scaling tended to improve as one moved down the list with the notable exception of the random toggling which performed rather worse than expected.

Table 2.6: **Fast Plane Rotation Algorithms with Conditions of Applicability**

	Standard	**Self-Scaling**	
		$\lvert d_p\rvert \geq \lvert d_q\rvert$	$\lvert d_p\rvert < \lvert d_q\rvert$
$\lvert\theta\rvert \leq \frac{\pi}{4}$ $\equiv$ $\frac{y_{q1}^2}{y_{p1}^2} \leq \frac{d_p^2}{d_q^2}$	$\alpha \Leftarrow y_{q1}/y_{p1}$ $\gamma = d_p^2/d_q^2$ $\beta \Leftarrow \alpha/\gamma$ $\delta = 1 + \beta * \alpha$ $F_{pq} = \begin{bmatrix} 1 & \beta \\ -\alpha & 1 \end{bmatrix}$ $\begin{Bmatrix} d^{2'}_p \Leftarrow d_p^2/\delta \\ d^{2'}_q \Leftarrow d_q^2/\delta \end{Bmatrix}$ $r \Leftarrow y_{p1} * \delta$	$\gamma = d_p^2/d_q^2$ $\tau = y_{q1}/y_{p1}$ $\beta \Leftarrow \tau/\gamma$ $\delta = 1 + \beta * \tau$ $\alpha \Leftarrow \tau/\delta$ $F_{pq} = \begin{bmatrix} 1 & 0 \\ -\alpha & 1 \end{bmatrix} \begin{bmatrix} 1 & \beta \\ 0 & 1 \end{bmatrix}$ $\begin{Bmatrix} d^{2'}_p \Leftarrow d_p^2/\delta \\ d^{2'}_q \Leftarrow d_q^2 * \delta \end{Bmatrix}$ $r \Leftarrow y_{p1} * \delta$	$\alpha \Leftarrow y_{q1}/y_{p1}$ $\gamma = d_p^2/d_q^2$ $\tau = \alpha/\gamma$ $\delta = 1 + \alpha * \tau$ $\beta \Leftarrow \tau/\delta$ $F_{pq} = \begin{bmatrix} 1 & \beta \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -\alpha & 1 \end{bmatrix}$ $\begin{Bmatrix} d^{2'}_p \Leftarrow d_p^2 * \delta \\ d^{2'}_q \Leftarrow d_q^2/\delta \end{Bmatrix}$ $r \Leftarrow y_{p1}$
$\lvert\theta\rvert > \frac{\pi}{4}$ $\equiv$ $\frac{y_{q1}^2}{y_{p1}^2} > \frac{d_p^2}{d_q^2}$	$\alpha \Leftarrow y_{p1}/y_{q1}$ $\gamma = d_p^2/d_q^2$ $\beta \Leftarrow \alpha * \gamma$ $\delta = 1 + \beta * \alpha$ $F_{pq} = \begin{bmatrix} \beta & 1 \\ -1 & \alpha \end{bmatrix}$ $\begin{Bmatrix} d^{2'}_p \Leftarrow d_q^2/\delta \\ d^{2'}_q \Leftarrow d_p^2/\delta \end{Bmatrix}$ $r \Leftarrow y_{q1} * \delta$	$\alpha \Leftarrow y_{p1}/y_{q1}$ $\gamma = d_p^2/d_q^2$ $\tau = \alpha * \gamma$ $\delta = 1 + \alpha * \tau$ $\beta \Leftarrow \tau/\delta$ $F_{pq} = \begin{bmatrix} 1 & -\beta \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ -1 & \alpha \end{bmatrix}$ $\begin{Bmatrix} d^{2'}_p \Leftarrow d_q^2 * \delta \\ d^{2'}_q \Leftarrow d_p^2/\delta \end{Bmatrix}$ $r \Leftarrow y_{q1}$	$\gamma = d_p^2/d_q^2$ $\tau = y_{p1}/y_{q1}$ $\beta \Leftarrow \tau * \gamma$ $\delta = 1 + \beta * \tau$ $\alpha \Leftarrow \tau/\delta$ $F_{pq} = \begin{bmatrix} 1 & 0 \\ \alpha & -1 \end{bmatrix} \begin{bmatrix} \beta & 1 \\ 1 & 0 \end{bmatrix}$ $\begin{Bmatrix} d^{2'}_p \Leftarrow d_q^2/\delta \\ d^{2'}_q \Leftarrow d_p^2 * \delta \end{Bmatrix}$ $r \Leftarrow y_{q1} * \delta$

"=" denotes an intermediate value assignment, and "⇐" denotes a final value assignment.

Note: the squares of the diagonal elements are implicit.

2.3 Alternative Plane Rotation Formulae

2.3.1 Single Diagonal Normalization

To normalize one of the diagonal vector elements in the course a rotation, one must sacrifice some of the computational benefit of having a coupled pair of linked triads. In all of the following expressions, the chaining between column updates is maintained. But, because the second equation in the following expressions is no longer a triad, I label these rotations *half-fast*.

Normalize d_q:

$$[\tilde{x}_p, \tilde{x}_q] \equiv [y_p, y_q] \begin{bmatrix} 1 & -sd_p \\ \beta & cd_q \end{bmatrix} \begin{bmatrix} cd_p & 0 \\ 0 & 1 \end{bmatrix}$$

$$\left\{ \begin{array}{rcl} \tilde{d}_p & \Leftarrow & cd_p \\ \check{d}_q & \Leftarrow & 1 \\ & & \\ \tilde{y}_p & \Leftarrow & y_p + \beta y_q \\ \check{y}_q & \Leftarrow & c^{-1} d_q y_q - sd_p \tilde{y}_p \end{array} \right\} \tag{2.19}$$

Normalize d_p:

$$[\tilde{x}_p, \tilde{x}_q] \equiv [y_p, y_q] \begin{bmatrix} cd_p & -\alpha \\ sd_q & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & cd_q \end{bmatrix}$$

$$\left\{ \begin{array}{rcl} \tilde{d}_q & \Leftarrow & cd_q \\ \check{d}_p & \Leftarrow & 1 \\ & & \\ \tilde{y}_q & \Leftarrow & y_q + \beta y_p \\ \check{y}_p & \Leftarrow & c^{-1} d_p y_p - sd_q \tilde{y}_q \end{array} \right\} \tag{2.20}$$

If a diagonal vector element is instead scaled to the reciprocal of the cosine of the current rotation angle, the following expressions hold:

Rescale d_q to c^{-1}:

$$[\tilde{x}_p, \tilde{x}_q] \equiv [y_p, y_q] \begin{bmatrix} 1 & -scd_p \\ \beta & c^2 d_q \end{bmatrix} \begin{bmatrix} cd_p & 0 \\ 0 & c^{-1} \end{bmatrix}$$

$$\left\{ \begin{array}{lcl} \tilde{d}_p & \Leftarrow & cd_p \\ \breve{d}_q & \Leftarrow & c^{-1} \\ & & \\ \tilde{y}_p & \Leftarrow & y_p + \beta y_q \\ \breve{y}_q & \Leftarrow & d_q y_q - scd_p \tilde{y}_p \end{array} \right\} \tag{2.21}$$

Rescale d_p to c^{-1}:

$$[\tilde{x}_p, \tilde{x}_q] \equiv [y_p, y_q] \begin{bmatrix} c^2 d_p & -\alpha \\ scd_q & 1 \end{bmatrix} \begin{bmatrix} c^{-1} & 0 \\ 0 & cd_q \end{bmatrix}$$

$$\left\{ \begin{array}{lcl} \tilde{d}_q & \Leftarrow & cd_q \\ \breve{d}_p & \Leftarrow & c^{-1} \\ & & \\ \tilde{y}_q & \Leftarrow & y_q - \alpha y_p \\ \breve{y}_p & \Leftarrow & d_p y_p + scd_q \tilde{y}_q \end{array} \right\} \tag{2.22}$$

And similarly, Rescale d_p to c with d_q increasing:

$$[\tilde{x}_p, \tilde{x}_q] \equiv [y_p, y_q] \begin{bmatrix} dp & -c^2\alpha \\ \beta d_p & c^2 \end{bmatrix} \begin{bmatrix} c & 0 \\ 0 & c^{-1} d_q \end{bmatrix}$$

$$
\left\{
\begin{array}{lcl}
\hat{d}_p & \Leftarrow & c \\
\breve{d}_q & \Leftarrow & c^{-1} d_q \\
 & & \\
\hat{y}_p & \Leftarrow & d_p y_p + \beta d_p y_q \\
\breve{y}_q & \Leftarrow & y_q - s c d_q^{-1} \hat{y}_p
\end{array}
\right\} \tag{2.23}
$$

Rescale d_q to c with d_p increasing:

$$
[\tilde{x}_p, \tilde{x}_q] \equiv [y_p, y_q]
\begin{bmatrix} c^2 & -\alpha d_p \\ c^2 \beta & d_q \end{bmatrix}
\begin{bmatrix} c^{-1} d_p & 0 \\ 0 & c \end{bmatrix}
$$

$$
\left\{
\begin{array}{lcl}
\hat{d}_q & \Leftarrow & c \\
\breve{d}_p & \Leftarrow & c^{-1} d_p \\
 & & \\
\hat{y}_q & \Leftarrow & d_q y_q - d_q \alpha y_p \\
\breve{y}_p & \Leftarrow & y_p + s c d_q^{-1} \hat{y}_q
\end{array}
\right\} \tag{2.24}
$$

and, for large angles, Normalize d_q:

$$
[\tilde{x}_p, \tilde{x}_q] \equiv [y_p, y_q]
\begin{bmatrix} \beta^{-1} & -s d_p \\ 1 & c d_q \end{bmatrix}
\begin{bmatrix} s d_q & 0 \\ 0 & 1 \end{bmatrix}
$$

$$
\left\{
\begin{array}{lcl}
\tilde{d}_p & \Leftarrow & s d_q \\
\breve{d}_q & \Leftarrow & 1 \\
 & & \\
\tilde{y}_p & \Leftarrow & y_q + \beta^{-1} y_p \\
\breve{y}_q & \Leftarrow & c d_q \tilde{y}_p - s^{-1} d_p y_p
\end{array}
\right\} \tag{2.25}
$$

Normalize d_p:

$$[\tilde{x}_p, \tilde{x}_q] \equiv [y_p, y_q] \begin{bmatrix} cd_p & 1 \\ sd_q & -\alpha^{-1} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & sd_p \end{bmatrix}$$

$$\left\{ \begin{array}{rcl} \tilde{d}_q & \Leftarrow & sd_p \\ \check{d}_p & \Leftarrow & 1 \\ & & \\ \tilde{y}_q & \Leftarrow & y_p - \alpha^{-1} y_q \\ \check{y}_p & \Leftarrow & s^{-1} d_q y_q + cd_p \tilde{y}_q \end{array} \right\} \tag{2.26}$$

Again, if a diagonal vector element is instead scaled to the reciprocal of the sine of the current rotation angle, the following expressions hold for the large angle formulations:

Rescale d_q to s^{-1}:

$$[\tilde{x}_p, \tilde{x}_q] \equiv [y_p, y_q] \begin{bmatrix} \beta^{-1} & -s^2 d_p \\ 1 & csd_q \end{bmatrix} \begin{bmatrix} sd_q & 0 \\ 0 & s^{-1} \end{bmatrix}$$

$$\left\{ \begin{array}{rcl} \tilde{d}_p & \Leftarrow & sd_q \\ \breve{d}_q & \Leftarrow & s^{-1} \\ & & \\ \tilde{y}_p & \Leftarrow & y_q + \beta^{-1} y_p \\ \breve{y}_q & \Leftarrow & csd_q \tilde{y}_p - d_p y_p \end{array} \right\} \tag{2.27}$$

Rescale d_p to s^{-1}:

$$[\tilde{x}_p, \tilde{x}_q] \equiv [y_p, y_q] \begin{bmatrix} csd_p & 1 \\ s^2 d_q & -\alpha^{-1} \end{bmatrix} \begin{bmatrix} s^{-1} & 0 \\ 0 & sd_p \end{bmatrix}$$

$$
\left\{
\begin{array}{lcl}
\tilde{d}_q & \Leftarrow & s d_p \\
\breve{d}_p & \Leftarrow & s^{-1} \\
 & & \\
 & & \\
\tilde{y}_q & \Leftarrow & y_p - \alpha^{-1} y_q \\
\breve{y}_p & \Leftarrow & d_q y_q + cs d_p \tilde{y}_q
\end{array}
\right\} \tag{2.28}
$$

And similarly, Rescale d_p to s with d_q set to an augmented d_p:

$$
[\tilde{x}_p, \tilde{x}_q] \equiv [y_p, y_q] \begin{bmatrix} \beta^{-1} d_q & s^2 \\ d_q & -s^2 \alpha^{-1} \end{bmatrix} \begin{bmatrix} s & 0 \\ 0 & s^{-1} d_p \end{bmatrix}
$$

$$
\left\{
\begin{array}{lcl}
\tilde{d}_p & \Leftarrow & s \\
\breve{d}_q & \Leftarrow & s^{-1} d_p \\
 & & \\
 & & \\
\tilde{y}_p & \Leftarrow & d_q y_q + \beta^{-1} d_q y_p \\
\breve{y}_q & \Leftarrow & y_p - cs d_p^{-1} \tilde{y}_p
\end{array}
\right\} \tag{2.29}
$$

Rescale d_q to s with d_p set to an augmented d_q:

$$
[\tilde{x}_p, \tilde{x}_q] \equiv [y_p, y_q] \begin{bmatrix} s^2 \beta^{-1} & d_p \\ s^2 & -\alpha^{-1} d_p \end{bmatrix} \begin{bmatrix} s^{-1} d_q & 0 \\ 0 & s \end{bmatrix}
$$

$$
\left\{
\begin{array}{lcl}
\tilde{d}_q & \Leftarrow & s \\
\breve{d}_p & \Leftarrow & s^{-1} d_q \\
 & & \\
 & & \\
\tilde{y}_q & \Leftarrow & d_p y_p - \alpha^{-1} d_p y_q \\
\breve{y}_p & \Leftarrow & y_q + cs d_p^{-1} \tilde{y}_q
\end{array}
\right\} \tag{2.30}
$$

2.3.2 Diagonal Pair Normalization

To normalize both of the diagonal vector elements in the course a rotation, the computation essentially requires a standard (not fast) plane rotation. However it is possible to chain the two column updates thus obviating the temporary vector store inherent in the standard rotation. In all of the following expressions, the chaining between column updates is maintained. The following expressions are applicable regardless of the angle magnitude.

Normalize d_p and d_q, with the y_p column update first:

$$[\tilde{x}_p, \tilde{x}_q] \equiv [y_p, y_q] \begin{bmatrix} cd_p & -sd_p \\ sd_q & cd_q \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\left\{ \begin{array}{lcl} \check{d}_p & \Leftarrow & 1 \\ \check{d}_q & \Leftarrow & 1 \\ & & \\ \check{y}_p & \Leftarrow & cd_p y_p + sd_q y_q \\ \check{y}_q & \Leftarrow & c^{-1} d_q y_q - t\check{y}_p \end{array} \right\} \tag{2.31}$$

Or, one can reverse the order of the chaining. Normalize d_p and d_q, with the y_q column update first:

$$\left\{ \begin{array}{lcl} \check{d}_q & \Leftarrow & 1 \\ \check{d}_p & \Leftarrow & 1 \\ & & \\ \check{y}_q & \Leftarrow & cd_q y_q - sd_p y_p \\ \check{y}_p & \Leftarrow & c^{-1} d_p y_p + t\check{y}_q \end{array} \right\} \tag{2.32}$$

If the cosine is too small, elements of the vector divided by the cosine may overflow. To prevent this, chaining must be eliminated, leaving a standard plane rotation with

normalization of the diagonal elements.

$$
\left\{
\begin{array}{lcl}
\check{d}_p & \Leftarrow & 1 \\
\check{d}_q & \Leftarrow & 1 \\
 & & \\
\check{y}_p & \Leftarrow & c d_p y_p + s d_q y_q \\
\check{y}_q & \Leftarrow & c d_q y_q - s d_p y_p
\end{array}
\right\} \tag{2.33}
$$

Chapter III

APPLICATIONS

3.1 Orthogonal Factorizations

3.1.1 QR Factorizations

Because the 2-norm remains invariant under orthogonal transformation, the Householder, modified Gram-Schmidt, and Givens orthogonal factorization (or decomposition) methods are commonly used for the solution of least squares problems (LS) and other problem application areas whose solution processes are facilitated by operations on specific structured decompositions. E.g., the factorization of a general matrix into the product of an orthogonal matrix and an upper triangular matrix, a QR factorization, allows for very efficient solution methods based on the special structure of the upper triangular matrix. The orthogonal factorization additionally yields beneficial numerical stability properties. [1]

[1] As an important implementation detail, if matrices are stored in row-major form, the $X = QR$ factorization should be performed, else, if storage is in column-major form, the $X^T = LQ^T$ factorization should be performed so as to promote and maximize contiguous data reference. Herein we will assume row-major storage and discuss the QR rather than the LQ^T factorization.

3.1.1.1 Householder QR Factorization

The Householder algorithm, which has many implementation variants, proceeds by zeroing all subdiagonal elements of successive columns (or sets of columns when blocked) at each iteration to form R. The orthogonal factor is constructed as a product of Householder reflection matrices, which are rank-1 modifications to the identity,

$$Q = Q_1 Q_2 \dots Q_n, Q_i = I - 2\frac{v_i v_i^T}{v_i^T v_i}, \tag{3.1}$$

where each v_i is chosen so that Q_i will zero the subdiagonal elements of the i'th column. The Q_i matrices have a special structure which should be exploited in computation to eliminate an order of magnitude of computational complexity. The algorithmic details are outlined in [GVL89]. Although Householder proposed using Householder transformations to solve (LS), the practical details were developed by Golub, et al. [GVL89].

3.1.1.2 Modified Gram-Schmidt QR Factorization

The Gram-Schmidt algorithm successively generates columns of Q which form an orthogonal basis for X where $X = QR$. At the i'th step of the method, the i'th column of both Q and R are generated. The Modified Gram-Schmidt algorithm is a more computationally stable variant of the Gram-Schmidt algorithm. At the i'th step of the MGS method, the i'th column of Q and the i'th *row* of R are generated. Again, the details are outlined in [GVL89]. Björck in [Bjö67] carried out a detailed error analysis of the MGS algorithm and applied it to the (LLS) problem. For further descriptions of the Householder and modified Gram-Schmidt methods, see [GVL89]. In [Bjö91], it is shown that MGS is numerically equivalent to the Householder method

applied to the a matrix with the $n \times n$ zero matrix adjoined to the top. Wampler [Wam79a, Wam79b] applies iterative refinement to MGS for the solution of the WLS problem.

3.1.1.3 Givens QR Factorization

The Givens QR factorization algorithm (as described in [GVL89]) is an algorithm for computing the factorization of a matrix $A \in \mathbb{R}^{m \times n}, m \geq n$, into the product of an orthogonal matrix, $Q \in \mathbb{R}^{m \times m}$, and an upper triangular matrix, $R \in \mathbb{R}^{m \times n}$, where Q is the product of a sequence of circular plane, i.e., Givens rotations:

$$Q = G_1 G_2 \ldots G_k, Q^T X = R,$$

where k equals the count of the subdiagonal elements; or equivalently,

$$Q \equiv \prod G(\theta, p, q), \forall p, q, p < q,$$

(for any ordering which does not destroy zeros). Each rotation, G_i satisfies the identity, $G_i^T G_i = I$, and is used to annihilate elements individually at each algorithmic step. A plane rotation $G(\theta, p, q)$ of order n through an angle θ in a (p, q) plane is the same as the identity matrix I_n, except for the four elements at the intersections of the p-th and q-th rows and columns. The general form of an $n \times n$ rotation matrix $G(\theta, p, q)$ in the (p, q) plane, which we will denote as G, is

$$G = G(\theta, p, q) = \begin{bmatrix} I_{p-1} & 0 & 0 & 0 & 0 \\ 0 & c & 0 & s & 0 \\ 0 & 0 & I_{q-p-1} & 0 & 0 \\ 0 & -s & 0 & c & 0 \\ 0 & 0 & 0 & 0 & I_{n-q} \end{bmatrix}, \tag{3.2}$$

where $c \equiv \cos\theta, s \equiv \sin\theta$. We will use the notations X_{pq} and G_{pq} to denote the 2×2 submatrices of X and $G(\theta, p, q)$ in the (p, q) plane, respectively, i.e.,

$$X_{pq} = \begin{bmatrix} x_{pp} & x_{pq} \\ x_{qp} & x_{qq} \end{bmatrix} \text{ and } G_{pq} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}.$$

It is well known that a Givens rotation can annihilate a specific element in a matrix as

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix}, \quad r = \sqrt{a^2 + b^2}, \quad c = a/r, \ and \quad s = -b/r. \tag{3.3}$$

The transposed equation is

$$\begin{bmatrix} r \\ 0 \end{bmatrix} \Leftarrow \begin{bmatrix} c & -s \\ s & c \end{bmatrix}^T \begin{bmatrix} a \\ b \end{bmatrix}.$$

Which is used depends on whether row-wise or column-wise operations are preferred. In practice, these values of $\cos\theta$, $\sin\theta$, (and implicitly θ) are reformulated to minimize overflow and roundoff error [GVL89] as shown in Algorithm 3.1.2. There are many orderings in which the elements can be annihilated. Some of these orderings promote the data locality and others permit many rows to be zeroed in parallel, readily admitting systolic implementations. (In practice, these equations are reformulated to minimize overflow and roundoff error. See [GVL89]) G^T rather than G may be used equivalently if care is taken to reflect the change in dependent calculations. There are many useful orderings of elements to be zeroed and their respective pivots. Some of these orderings promote the data locality of the rotated rows. Other orderings permit many rows to be zeroed in parallel. Furthermore, plane rotations readily admit systolic implementations.

A verified library of Givens QR factorization algorithms expressed in Matlab can be found in Appendix B. Fortran versions and variations of these algorithms were implemented and timed on a Cray C90. Refer to the Efficiency Chapter for the results.

3.1.2 Plane Rotation Based Factorizations

The plane rotation is commonly applied in several different algorithmic contexts. These algorithms differ in the method, purpose, and range of the angle computation.

(Note: this list is exemplary and is not intended to be an exhaustive list of algorithms to which the plane rotation are be applied. Other methods including the Paardekooper method, the Kogbetliantz method, and the Givens (EVD) Method also utilize plane rotations.)

The Givens QR Factorization algorithm is described in Section (3.1.1.3).

3.1.2.1 Hestenes' Method for the Singular Value Decomposition

Hestenes' method (as described in [BL85, dR89]) is an algorithm for computing the *singular value decomposition.* Given a matrix $A \in \mathbb{R}^{m \times n}, m \geq n$, the decomposition $A \equiv U\Sigma V^T$ is computed by orthogonalizing the columns of A, forming W, by post-multiplying A with a product of plane rotations which represent the right singular vectors, V, and further normalizing those columns of W to yield the diagonal matrix of singular values, Σ, and the matrix of left singular vectors, U.

$$\begin{aligned} A &\equiv W\left(\prod J\right) \\ &\equiv WV^T \\ &\equiv U\Sigma V^T \end{aligned}$$

The rotation angle, θ, of each J, which orthogonalizes the two columns, w_p and w_q of some intermediate matrix converging to W, is determined via Algorithm 3.1.2.1:

de Rijk [dR89] remarks that the rotation angles tend to zero during later sweeps. The indices p and q are usually selected cyclicly.

Algorithm 3.1 Givens rotation parameters for the QR method

INPUT: a, b; scalars, with b to be zeroed.

OUTPUT: $\sin\theta, \cos\theta$; rotation coefficients.

if $b = 0$ **then**

$\cos\theta \leftarrow 1$

$\sin\theta \leftarrow 0$

elseif $|b| > |a|$ **then**

$\tan\theta \leftarrow -a/b$

$\sin\theta \leftarrow 1/\sqrt{1+\tan^2\theta}$

$\cos\theta \leftarrow \sin\theta\tan\theta$

else

$\tan\theta \leftarrow -b/a$

$\cos\theta \leftarrow 1/\sqrt{1+\tan^2\theta}$

$\sin\theta \leftarrow \cos\theta\tan\theta$

endif

There is no bound on the value of θ.

3.1.2.2 Jacobi Eigenvalue and Eigenvector Decomposition

The Jacobi eigenvalue and eigenvector decomposition method is based on the Schur decomposition: Given a matrix $A \in \mathbb{R}^{n\times n}$, the decomposition,

$$A \equiv QRQ^{-1} \equiv QRQ^{*} \equiv \left(\prod J\right) R \left(\prod J^{*}\right)$$

Algorithm 3.2 Rotation parameters for the Hestenes method

INPUT: w_p, w_q; vectors to be orthogonalized.

OUTPUT: $\sin\theta, \cos\theta$; rotation coefficients.

$$\alpha \Leftarrow \|w_p\|_2^2$$

$$\beta \Leftarrow \|w_q\|_2^2$$

$$\gamma \Leftarrow w_p^T w_q$$

if $\gamma = 0$ **then**

$$\theta \equiv 0 \Longleftrightarrow \begin{cases} \cos\theta \Leftarrow 1 \\ \sin\theta \Leftarrow 0 \end{cases}$$

else

$$\xi \Leftarrow \frac{\beta-\alpha}{2\gamma}$$

$$\tan\theta \Leftarrow \frac{\text{sign}(\xi)}{|\xi|+\sqrt{1+\xi^2}}$$

$$\cos\theta \Leftarrow \frac{1}{\sqrt{1+\tan^2\theta}}$$

$$\sin\theta \Leftarrow \tan\theta \cdot \cos\theta$$

endif

The equation for $\tan\theta$ yields the following bound on θ: $|\theta| \leq \frac{\pi}{4}$.

---

where, the diagonal elements of the upper triangular matrix R are the eigenvalues of A, and the columns of the orthogonal matrix Q are the eigenvectors of A. If A is Hermitian, R is a diagonal real matrix. The superscript $*$ denotes the conjugate transpose.

The rotation angle, θ, of each J, which zeros the subdiagonal element (p, q) of some intermediate matrix converging to R, is determined via Algorithm 3.1.2.2:

Algorithm 3.3 Rotation parameters for the symmetric Jacobi EVD method

INPUT: $A^k_{pq}, A^k_{pp}, A^k_{qq}$; scalars, with A^k_{pq} element to be zeroed.

OUTPUT: $\sin\theta, \cos\theta$; rotation coefficients.

Let A^k_{pq} be the $(p,q)th$ element of some intermediate symmetric matrix A^k. (from [dR89])

if $A^k_{pq} \equiv 0$ **then**

$$\theta \equiv 0 \Longleftrightarrow \begin{cases} \cos\theta & \Leftarrow & 1 \\ \sin\theta & \Leftarrow & 0 \end{cases}$$

else

$$\begin{aligned} \xi &\Leftarrow \frac{A^k_{qq} - A^k_{pp}}{2A^k_{pq}} \\ \tan\theta &\Leftarrow \frac{\text{sign}(\xi)}{|\xi| + \sqrt{1+\xi^2}} \\ \cos\theta &\Leftarrow \frac{1}{\sqrt{1+\tan^2\theta}} \\ \sin\theta &\Leftarrow \tan\theta \cdot \cos\theta \end{aligned}$$

endif

The equation for $\tan\theta$ yields the following bound on θ: $|\theta| \leq \frac{\pi}{4}$.

Otherwise, in the non-Hermitian case, the algorithm is much more labyrinthine: complex conjugate sine pairs are required, and the choice of two possible θ's is dependent upon the data values, the data locations, and some *rule-of-thumb* threshold parameters. *In this case, θ is not bounded.* For a detailed discussion and statement of an algorithm for the non-Hermitian case, see [Ebe87].

3.2 Applied Jacobi Transformations

Jacobi algorithms for the algebraic eigenvalue decomposition have gained special attention recently due to their ability to produce the eigenvalues and eigenvectors with higher accuracy in addition to their excellent adaptability for vector and parallel computing environments [DV88, EP90]. See Appendix D for Matlab programs and results using an example matrix from [DV88]. But, Jacobi algorithms, compared to QR algorithms, have the disadvantage that they require greater computational costs. The computational kernel in a Jacobi algorithm is the plane rotation. Although it has been well known that a substantial reduction in computational costs can be achieved by using fast rotations, their use has been avoided due primarily to the possibility of underflow and overflow. Several methods for resolving this problem, such as periodic rescaling, had been suggested in the past but their practicality in advanced computing environments has been doubtful [GVL89].

Reductions in computation cost due to communication delays may often be accomplished in Jacobi algorithms by replacing two-sided methods by their one-sided variations. In the one-sided variations of Jacobi algorithms, we update only columns of the matrix, accumulate the rotations if necessary, and leave the result of each stage in a factored form. We then retrieve the matrix elements only when necessary, instead of explicitly applying the rotations from the left side of the matrix. Accordingly, the communication cost for the rotation parameter passing is substantially reduced on message-passing systems. Also, on a vector processor, higher efficiency is expected since one-sided updates avoid vector operations with non-unit strides.

3.3 One-Sided Jacobi Algorithms

A unitary plane rotation $R(p,q)$ of order n in the (p,q) plane is the same as the identity matrix I_n, except for the four elements at the intersections of the p-th and q-th rows and columns. The general form of an $n \times n$ rotation matrix $R(p,q)$ in the (p,q) plane, which we will denote as R, is

$$R = R(p,q) = \begin{bmatrix} I_{p-1} & 0 & 0 & 0 & 0 \\ 0 & c & 0 & -\bar{s} & 0 \\ 0 & 0 & I_{q-p-1} & 0 & 0 \\ 0 & s & 0 & c & 0 \\ 0 & 0 & 0 & 0 & I_{n-q} \end{bmatrix}, \tag{3.4}$$

where $c \equiv \cos x$, $s \equiv e^{-i\theta} \sin x$, x and $\theta \in \mathbb{R}$, and $\bar{s}$ denotes the complex conjugate of s. Plane rotations constitute a kernel on which Jacobi algorithms are based, and the same is the case for other reduction methods in numerical linear algebra such as the QR factorization, the QR algorithms for eigendecompositions, the Hestenes algorithm for the singular value decomposition, and the reduction to Hessenberg form. Although these algorithms differ in the purpose and range of the angles, we can classify them largely into two categories: those algorithms which apply two-sided transformations,

$$\hat{X} = J(p,q)^* X R(p,q), \tag{3.5}$$

where $J(p,q)$ is also a unitary plane rotation and the superscript $*$ denotes the conjugate transpose, and other algorithms which apply one-sided transformations,

$$\hat{X} = X R(p,q). \tag{3.6}$$

These Jacobi methods are applicable in both real and complex contexts. We will use the notation X_{pq} to denote the 2×2 submatrix of X in the (p, q) plane, i.e.,

$$X_{pq} = \begin{bmatrix} x_{pp} & x_{pq} \\ x_{qp} & x_{qq} \end{bmatrix}.$$

A typical step of a Jacobi-type algorithm for a matrix $X \in \mathbb{C}^{n \times n}$ computes a transformation,

$$\tilde{X} = J(p, q)^* X R(p, q), \tag{3.7}$$

to annihilate certain off-diagonal elements in X_{pq}. The plane rotations J and R may be chosen in various ways to obtain appropriate decompositions. For the Hermitian eigenvalue decomposition, the choice of a rotation in the (p, q) plane, J, equal to R, allows us to annihilate x_{pq} and x_{qp}, and for the non-Hermitian eigenvalue decomposition, to annihilate x_{qp}. For the Kogbetliantz singular value decomposition algorithm, we can choose two unitary rotations, J and R, to reduce X_{pq} to a diagonal form. In the one-sided variations of the Jacobi algorithms, we update columns of the matrix, accumulate the rotations if necessary, and retrieve the matrix elements that are required for the computation of the rotations. [EP90, PE90, VH89]. Thus the rotations are never explicitly applied from the left side of a matrix. Suppose a two-sided Jacobi algorithm generates $X^{(k+1)}$ at the k-th stage, via

$$X^{(k+1)} = J^{(k)^*} X^{(k)} R^{(k)}, \tag{3.8}$$

with $X^{(1)} = X$. Then in its one-sided variation, we generate a matrix pair, $(A^{(k)}, U^{(k)})$, where

$$A^{(k+1)} = A^{(k)} R^{(k)} \text{ and} \tag{3.9}$$

$$U^{(k+1)} = U^{(k)} J^{(k)}, \tag{3.10}$$

with $A^{(1)} = X$, and $U^{(1)} = I$. Then, at any stage k, the product, $U^{(k)^*} A^{(k)}$ gives $X^{(k)}$ since

$$\begin{aligned} U^{(k)^*} A^{(k)} = \left(\prod_{l=1}^{k-1} J^{(l)} \right)^* A^{(1)} \left(\prod_{l=1}^{k-1} R^{(l)} \right) &= J^{(k-1)^*} X^{(k-1)} R^{(k-1)} \\ &= X^{(k)}, \end{aligned} \tag{3.11}$$

and from Eqn. (8), we can recover the 2×2 submatrix of $X^{(k)}$ necessary to compute the plane rotations:

$$x_{ij}^{(k)} = \left\langle u_i^{(k)}, a_j^{(k)} \right\rangle, \text{ for } \{i, j\} = \{p, q\}, \tag{3.12}$$

where $\langle \cdot, \cdot \rangle$ denotes the inner product, and the subscript denotes the corresponding column. At the expense of storing an additional vector for the diagonal elements of X, the diagonal elements x_{pp} and x_{qq} can be updated explicitly by simple arithmetic, and the number of inner products can be reduced. Thus, row updates are eliminated at a cost of one or two extra dot products each time a rotation is computed. Note that unlike in the two-sided algorithms, the unitary matrix $J^{(k)}$ of each stage k must be accumulated in order to recover the necessary elements of the matrix $X^{(k)}$ [EP90, PE90]. When the matrix is Hermitian positive definite, an initial Cholesky decomposition can make the one-sided method even simpler [VH89]. Let L be the lower-triangular Cholesky factor of X, i.e., $X = LL^*$, and define $M = L^*L$. If Z is a unitary matrix satisfying

$$Z^* M Z = \Lambda = \text{diag}(\lambda_1, \ldots, \lambda_n), \tag{3.13}$$

then $\{\lambda_1, \ldots, \lambda_n\}$ are the eigenvalues of X since $M = L^{-1}XL$. Also $S \equiv LZ\Lambda^{-1/2}$ is unitary and we have

$$S^*XS = \Lambda^{-1/2}\Lambda^2\Lambda^{-1/2} = \Lambda. \tag{3.14}$$

Thus, the one-sided algorithm that computes $L^{(k+1)} = L^{(k)}R^{(k)}$ at each stage k, where $R^{(k)}$ is chosen to annihilate the off diagonal elements in a 2×2 submatrix of $R^{(k)^*}L^{(k)^*}L^{(k)}R^{(k)}$, with $L^{(1)} = L$, gives the eigenvectors, which are the normalized columns of $L^{(k)}$ after convergence. The diagonal elements of $R^{(k)^*}L^{(k)^*}L^{(k)}R^{(k)}$ can be stored separately and updated explicitly each time to give the eigenvalues upon convergence. Thus, no extra rotation accumulation, as in Eqn. (3.10), is needed. If X is ill-conditioned with respect to inversion, there may be accuracy problems which can be solved by carefully performing the initial Cholesky decomposition, e.g., using arithmetic of higher precision. For more details on one-sided algorithms, see [EP90, PE90, PH93, VH89].

Since any two-sided Jacobi algorithm can be modified into a one-sided algorithm for better vectorization and parallelization, as shown in the previous section, we will use fast rotations for column oriented one-sided transformations. To simplify the discussion further, we will restrict the two-sided transformations to orthogonal similarity transformations. For complex fast rotations, see [PH93]. When the left and right rotation matrices are different, each of them needs to be transformed to a fast rotation separately. The results for row oriented and two-sided transformations follow trivially.

3.4 Row Modifications

Often numerical problems occur in sequences such that after a factorization and solution is effected, a subsequent problem is to be solved where the subsequent matrix and data are modified only slightly from the previous. We will examine the special cases where the difference between the previous and subsequent matrices is such that a row or column is adjoined or deleted. Specifically, we will consider only when rows are added (*updating*), and when rows are deleted (*downdating*). Let us first examine the case where a row is added.

3.4.1 Updating

The matrix C is updated with the row v^T by forming the augmented matrix

$$\check{C} = \begin{bmatrix} C \\ v^T \end{bmatrix}. \tag{3.15}$$

The location of the row insertion will have no effect for the least squares update. The updating of the Cholesky factor R of C^TC to form the Cholesky factor $\check{R}$ for $\check{C}^T\check{C}$ satisfies the equation,

$$\check{C}^T\check{C} = C^TC + vv^T = R^TR + vv^T = \check{R}^T\check{R}. \tag{3.16}$$

The updated factor, $\check{R}$, is generated by an orthogonal transformation

$$\begin{bmatrix} \check{R} \\ 0^T \end{bmatrix} = \check{J} \begin{bmatrix} R \\ v^T \end{bmatrix}. \tag{3.17}$$

This is demonstrated by the augmented matrix formulation of equation (3.16),

$$\check{R}^T\check{R} \;=\; R^TR + vv^T$$

$$
\begin{aligned}
&= \begin{bmatrix} R \\ v^T \end{bmatrix}^T \begin{bmatrix} R \\ v^T \end{bmatrix} \\
&= \begin{bmatrix} R \\ v^T \end{bmatrix}^T \begin{bmatrix} I_n & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R \\ v^T \end{bmatrix} \\
&= \begin{bmatrix} R \\ v^T \end{bmatrix}^T J^T \begin{bmatrix} I_n & 0 \\ 0 & 1 \end{bmatrix} J \begin{bmatrix} R \\ v^T \end{bmatrix} \\
&= \begin{bmatrix} \check{R} \\ 0^T \end{bmatrix}^T \begin{bmatrix} I_n & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \check{R} \\ 0^T \end{bmatrix},
\end{aligned}
$$

(3.18)

where J must satisfy the equation, $J^T I_{n+1} J = I_{n+1}$. A matrix J which satisfies this equation is orthogonal. $\check{J}$ is actually the product of a sequence of Givens transformations which successively nullify the leftmost nonzero element of each successive partial transformation of the row vector v^T:

$$
\check{J} = \prod_{i=1}^{n} G(\theta, i, n+1), \tag{3.19}
$$

where each Givens transformations may be defined as

$$
G(\theta, p, q) = \begin{bmatrix}
I_{p-1} & 0 & 0 & 0 & 0 \\
0 & c & 0 & -s & 0 \\
0 & 0 & I_{q-p-1} & 0 & 0 \\
0 & s & 0 & c & 0 \\
0 & 0 & 0 & 0 & I_{n-q}
\end{bmatrix}, \tag{3.20}
$$

where $c = \cos(\theta)$, and $s = \sin(\theta)$, for some θ. This definition of a Givens transformation represents a rotation, though any Givens matrix, i.e. any member of the

real orthogonal group of 2×2 matrices, will suffice, e.g. the Givens reflection matrix $\begin{bmatrix} c & s \\ s & -c \end{bmatrix}$.

3.4.2 Downdating

Assuming that the matrices $C$and $\tilde{C}$ have full rank, C is downdated by the removal of row v^T:

$$C = \begin{bmatrix} \tilde{C} \\ v^T \end{bmatrix}, \tag{3.21}$$

where $\tilde{C}$ represents the rows of C remaining after the deletion. This implies that the following equation holds:

$$\tilde{C}^T\tilde{C} = C^TC - vv^T = R^TR - vv^T = \tilde{R}^T\tilde{R}, \tag{3.22}$$

where R, and $\tilde{R}$ are the Cholesky factors of C^TC and $\tilde{C}^T\tilde{C}$, respectively. The downdate, however, cannot be effected by simply applying orthogonal transformations on R, as was the case for the update. This is evident in the augmented matrix representation,

$$\begin{aligned} \tilde{R}^T\tilde{R} &= R^TR - vv^T \\ &= \begin{bmatrix} R \\ v^T \end{bmatrix}^T \begin{bmatrix} I_n & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} R \\ v^T \end{bmatrix} \\ &= \begin{bmatrix} R \\ v^T \end{bmatrix}^T H^T \begin{bmatrix} I_n & 0 \\ 0 & -1 \end{bmatrix} H \begin{bmatrix} R \\ v^T \end{bmatrix} \\ &= \begin{bmatrix} \tilde{R} \\ 0^T \end{bmatrix}^T \begin{bmatrix} I_n & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} \tilde{R} \\ 0^T \end{bmatrix}, \end{aligned} \tag{3.23}$$

where the H that satisfies the equation,

$$H^T \begin{bmatrix} I_n & 0 \\ 0 & -1 \end{bmatrix} H = \begin{bmatrix} I_n & 0 \\ 0 & -1 \end{bmatrix}, \tag{3.24}$$

is not orthogonal. Several methods have been developed for either reformulating the problem so that orthogonal transformation can be used or determining the nature of the H transformations. Downdating methods which employ a reformulation to permit the use of orthogonal transformations are the *LINPACK* methods and their derivatives such as *Pan's* method. Downdating methods which directly determine and apply H are the *hyperbolic* methods.

Additionally, there are other downdating algorithms which correspond to other methods of factorizing and solving the linear least squares problem. Two other methods which are becoming relatively prominent in the literature are downdating via CSNE and downdating via MGS. [DGKS76, BPE94]

3.4.2.1 LINPACK Downdating

The most established and analyzed downdating method is that employed in LINPACK as described in [DBMS78]. This method is based on the following augmented system reformulation and factorization.

$$\begin{aligned} QC &= \begin{bmatrix} R \\ 0^T \end{bmatrix} \\ C &= Q^T \begin{bmatrix} R \\ 0 \end{bmatrix} \end{aligned}$$

$$\begin{bmatrix} \tilde{C} \\ v^T \end{bmatrix} = \begin{bmatrix} Q_1^T & u & Q_2^T \\ p^T & \alpha & q^T \end{bmatrix} \begin{bmatrix} R \\ 0^T \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} \tilde{C} \\ v^T \end{bmatrix} = \begin{bmatrix} Q_1^T & \hat{u} & \hat{Q}_2^T \\ p^T & \hat{\alpha} & 0^T \end{bmatrix} \begin{bmatrix} R \\ 0^T \\ 0 \end{bmatrix}.$$

This shows that p and $\hat{\alpha}$ are derivable from v and R, i.e., $v^T = p^T R$, or transposing, $R^T p = v$. And, $\hat{\alpha}^2 = 1 - p^T p$. The vector, p is found by forward-substitution to solve the triangular system, $R^T p = v$. Now, orthogonal transformations can be used to compute the downdated $\tilde{R}$:

$$\tilde{Q} \begin{bmatrix} p & R \\ \hat{\alpha} & 0^T \end{bmatrix} = \begin{bmatrix} 0 & \tilde{R} \\ \pm 1 & \pm v^T \end{bmatrix}. \tag{3.25}$$

The scalar, $p^T p$, is a good indicator of the condition of the system. If $1 - p^T p \ll 1$, $\hat{\alpha}$ will be computed with less accuracy, resulting in a less accurate $\tilde{R}$. [LH74, EP94]

C. -T. Pan [Pan90] modifies the LINPACK downdating algorithm by interleaving the triangular solve with the orthogonal transformations.

3.4.3 Hyperbolic Downdating

3.4.3.1 Pseudo-orthogonality

A two dimensional hyperbolic transformation matrix H is defined by the following relation:

$$H^T \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} H = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{3.26}$$

In a number of contemporary articles where hyperbolic-like transformations are derived and utilized, the authors generalize the hyperbolic transformation. Several similar labels and definitions have been offered:

pseudo-orthogonality [DI86] $\theta \in \mathbb{R}^{2m\times 2m}$ is *pseudo-orthogonal* iff

$$\theta^T \begin{bmatrix} I_m & 0 \\ 0 & -I_m \end{bmatrix} \theta = \begin{bmatrix} I_m & 0 \\ 0 & -I_m \end{bmatrix}. \tag{3.27}$$

J-orthogonality [CKLA87] θ is *J-orthogonal* iff $J = \theta J \theta^T$, where J is a diagonal matrix with elements being ± 1. The product of J-orthogonal matrices is J-orthogonal.

hyper-normality [RS86] The definition for *hyper-normality* is the same as that for J-orthogonality. Hyper-normal matrices preserve the hyperbolic norm of a vector, i.e., if $y^T = x^T\theta$, then $y^T J y = x^T J x$. Hyper-normal matrices also exhibit *hyperbolic symmetry*, where $\theta^T J \theta = \theta J \theta^T$. The hyperbolic norm of a complex column vector, x, with respect to J, is defined to be

$$x^T J x = \sum_i |x_i|^2 J_{ii}. \tag{3.28}$$

Σ-unitary [BBvd87] A matrix, A, is Σ-unitary iff

$$A^T \Sigma A = \Sigma, \Sigma = \begin{bmatrix} -1 & 0^T \\ 0 & I \end{bmatrix} \tag{3.29}$$

W-unitary [CB90] A matrix, U, is *W-unitary* iff $U^* W U = W$, for a $W = (w_{ij}), (w_{ii}) = \pm 1$, and W is Hermitian idempotent and unitary, i.e., $W^2 = I_n, W^* = W$. W-unitary matrices form a multiplicative group, i.e., I_n, products, and inverses of W-unitary matrices are W-unitary.

(J_1, J_2)-**unitary** [BG81] The nonsingular complex matrix H is (J_1, J_2)-unitary iff $H^* J_1 H = J_2$, where J_1, J_2 are $n \times n$ diagonal matrices with ± 1 entries on the diagonal with both matrices having exactly k negative entries.

3.4.3.2 Hyperbolic Householder Transformation

If a one set of rows is updated while another set is downdated, hyperbolic *Householder* transformations may be used. Hyperbolic Householder matrices are hypernormal. The problem is to find the lower triangular Cholesky factor $\hat{C}$ of S, where

$$S = X^T X + Y^T Y - Z^T Z, \tag{3.30}$$

and $X \in \mathbb{R}^{n \times n}$ is the original lower triangular factor, $Y \in \mathbb{R}^{l \times n}$ are the rows to be updated, and $Z \in \mathbb{R}^{p \times n}$ are the rows to be downdated. Let the diagonal matrix ϕ, the target of the hypernormal transformation, be defined with

$$\phi_{ii} = \begin{cases} 1 & i \leq n + l \\ -1 & n + l < i \leq n + l + p. \end{cases} \tag{3.31}$$

The normal Householder transformation for an arbitrary real matrix B has the form,

$$I - 2\frac{BB^T}{B^T B}, \tag{3.32}$$

but the hypernormal Householder transformation has the form,

$$\phi - 2\frac{BB^T}{B^T \phi B}. \tag{3.33}$$

The hyperbolic Householder algorithm is show in (Algorithm 3.4.3.2). [2] (The ϕ matrix is set as above)

---

[2] This algorithm is best used when $l + p + 2 < n/3$. [RS86]

Algorithm 3.4 Hyperbolic Householder

INPUT: $C = [X^T|Y^T|Z^T]^T$

OUTPUT: C (modified)

for $i = 1 : n$ **do**

$U \equiv (0, \ldots, 0, C_{i,i}, 0, \ldots, 0, C_{n+1,i}, \ldots, C_{n+l+p,1})^T$

$B \Leftarrow \phi U + \sigma e_i$, *where* $\sigma = (u_i/|u_i|)\sqrt{U^T \phi U}$

$Q \Leftarrow \phi - 2BB^T/(B^T \phi B)$

$C \Leftarrow QC$

endfor

3.4.3.3 Hyperbolic Rotations

One motivation for the development of hyperbolic algorithms is the elimination of the triangular forward-substitution of the LINPACK algorithm. There are three popular derivations for the hyperbolic rotation algorithm. Each derivation yields formulations identical with the other derivations, however, their origins are rather conceptually disparate.

The first derivation is that which gave rise to the label *hyperbolic*. [CKLA87] *Circular* rotations satisfy

$$G^T I G = G^T G = I. \tag{3.34}$$

For the 2×2 case, the equation $c^2 + s^2 = 1$ must be satisfied. The solution to this equation is for $c \equiv \cos\theta$ and $s \equiv \sin\theta$, for any θ. One valid circular rotation matrix

G has the structure,

$$G = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}. \tag{3.35}$$

For a valid matrix representation, the matrix G must have an odd number of prefixed negative signs, though they may occur in any location so there exist circular rotation *and* circular reflection matrices.

However, the pseudo-orthogonality conditions,

$$H^T \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} H = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \tag{3.36}$$

require the solution to the equation, $c^2 - s^2 = 1$. It was well known that one set of functions which satisfied this equation is for $c \equiv \cosh\vartheta$ and $s \equiv \sinh\vartheta$, for any ϑ. One valid hyperbolic rotation matrix H has the structure,

$$H = \begin{bmatrix} \cosh\vartheta & -\sinh\vartheta \\ -\sinh\vartheta & \cosh\vartheta \end{bmatrix}. \tag{3.37}$$

A valid matrix representation of H must be symmetric and have an even number of prefixed negative signs.

Either of the matrices G or H can be used to annihilate an element of vector. To annihilate the qth element of the vector $a^T G_{p,q}$ with respect to the pth element, set

$$\cos\theta = \frac{a_p}{\sqrt{a_p^2 + a_q^2}}, \qquad \sin\theta = \frac{a_q}{\sqrt{a_p^2 + a_q^2}}. \tag{3.38}$$

To annihilate the qth element of the vector $a^T H_{p,q}$ with respect to the pth element, set

$$\cosh\vartheta = \frac{a_p}{\sqrt{a_p^2 - a_q^2}}, \qquad \sinh\vartheta = \frac{a_q}{\sqrt{a_p^2 - a_q^2}}. \tag{3.39}$$

The second derivation is an algebraic reordering of the update equation. Rewriting equation (3.17),

$$\begin{bmatrix} R \\ 0^T \end{bmatrix} = G \begin{bmatrix} \tilde{R} \\ v^T \end{bmatrix}, \tag{3.40}$$

and can be found in [BBvd87]. The matrix G represents an orthogonal transformation. And define

$$\tilde{R}^T \tilde{R} = R^T R - v v^T \tag{3.41}$$

$$R^T R = \tilde{R}^T \tilde{R} + v v^T. \tag{3.42}$$

Expanding G into its elemental 2×2 factors,

$$\begin{bmatrix} R \\ 0^T \end{bmatrix} = G_n G_{n-1} \ldots G_2 G_1 \begin{bmatrix} \tilde{R} \\ v^T \end{bmatrix}, \tag{3.43}$$

define the ith product as

$$\begin{bmatrix} R_{i+1} \\ v_{i+1}^T \end{bmatrix} = G_i \begin{bmatrix} R_i \\ v_i^T \end{bmatrix}, \text{where} \tag{3.44}$$

$$R_1 \equiv \tilde{R}, \quad R_{n+1} \equiv R, \quad v_1^T \equiv v^T, \quad v_{n+1}^T \equiv 0^T.$$

For the downdate, find a matrix H_i such that

$$\begin{bmatrix} R_i \\ v_{i+1}^T \end{bmatrix} = H_i \begin{bmatrix} R_{i+1} \\ v_i^T \end{bmatrix}, \tag{3.45}$$

so that, if $H \equiv H_n H_{n-1} \ldots H_2 H_1$,

$$\begin{bmatrix} \tilde{R} \\ 0^T \end{bmatrix} = H \begin{bmatrix} R \\ v^T \end{bmatrix}. \tag{3.46}$$

Expanding equation (3.44),

$$\begin{bmatrix} R_{i+1} \\ v_{i+1}^T \end{bmatrix} = \begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} R_i \\ v_i^T \end{bmatrix} = \begin{bmatrix} cR_i & + & sv_i^T \\ -sR_i & + & cv_i^T \end{bmatrix}, \tag{3.47}$$

and rearranging,

$$\begin{aligned} cR_i &= R_{i+1} - sv_i^T \\ v_{i+1}^T &= -sR_i + cv_i^T, \end{aligned} \tag{3.48}$$

equivalently:

$$\begin{aligned} R_i &= \frac{1}{c}R_{i+1} - tv_i^T \\ v_{i+1}^T &= -s(\frac{1}{c}R_{i+1} - tv_i^T) + cv_i^T, \end{aligned} \tag{3.49}$$

$$\begin{aligned} R_i &= \frac{1}{c}R_{i+1} - tv_i^T \\ v_{i+1}^T &= -tR_{i+1} + (\frac{s^2}{c})v_i^T, \end{aligned} \tag{3.50}$$

and

$$\begin{aligned} R_i &= \frac{1}{c}R_{i+1} - tv_i^T \\ v_{i+1}^T &= -tR_{i+1} + \frac{1}{c}v_i^T, \end{aligned} \tag{3.51}$$

which has the matrix representation,

$$\begin{bmatrix} R_i \\ v_{i+1}^T \end{bmatrix} = \begin{bmatrix} \frac{1}{c} & -t \\ -t & \frac{1}{c} \end{bmatrix} \begin{bmatrix} R_{i+1} \\ v_i^T \end{bmatrix}. \tag{3.52}$$

The matrix

$$\begin{bmatrix} \frac{1}{c} & -t \\ -t & \frac{1}{c} \end{bmatrix} \tag{3.53}$$

satisfies the pseudo-orthogonality condition, $H^T J H = J$, and also, $(\frac{1}{c})^2 - t^2 = 1$. The ordered pair, $\{\sec\theta, \tan\theta\}$, is equivalent to $\{\cosh\vartheta, \sinh\vartheta\}$ if the unbounded independent variable, ϑ, is rescaled to the $\pm\frac{\pi}{2}$ bounded θ.

The third derivation, proposed in [Gol69], is based on the observation that if the same row is included in a least squares fit several times with various positive and negative weights, the net effect is as if it was included only once with the sum of the weights. G. Golub in [Gol69], proposed then that one may remove a row by adding the negative of the previous weight. For an unweighted problem, all weights are one, so a weight of -1 is used to downdate. In the factored form, the square root of the weight is used, so the downdate is effected by updating the product of $\sqrt{-1}$ with the row to be removed. [Gen73b] Upon inspection, the application of this idea to the update equations shows that all of the calculations are either in the pure real or the pure imaginary domain, so real arithmetic may be used throughout. The effective rotation matrix,

$$H = \begin{bmatrix} c & \sqrt{-1}s \\ -\sqrt{-1}s & c \end{bmatrix}, \tag{3.54}$$

also satisfies the pseudo-orthogonality condition $H^T J H = J$, and

$$c^2 + (\sqrt{-1}s)^2 = c^2 - s^2 = 1. \tag{3.55}$$

Note, however, that c and s are now computed as in Equation 3.39.

Although circular plane rotations have a condition of unity, the hyperbolic rotation has a condition,

$$\mathrm{cond}(H) = \frac{|c| + |s|}{|c| - |s|}, \tag{3.56}$$

which ultimately results in the potential instability of the downdating transformation. [EGK91]

3.5 Least Squares Problems with Equality Constraints

Least squares problems with equality constraints (LSE) may be represented as

$$\min_{Bx=d} \|Ax - b\|_2 , \tag{3.57}$$

where $A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{l \times n}, m \geq n$, and $n \geq l$. We will assume that $\text{rank}(B) = l$, and that $\text{null}(A) \bigcap \text{null}(B) = \{0\}$. If $\text{rank}\left(\begin{bmatrix} B \\ A \end{bmatrix}\right) < n$, a minimum norm solution can be specified.

We have previously presented self-scaling fast plane rotations [AP94a] which obviate the rescaling necessary in other fast plane rotations. In this section, we present algorithms that apply self-scaling fast plane rotations to the QR factorization for stiff least squares problems. These problems appear when an equality constrained linear least square problem is solved via extreme weighting of the constraint equations, for example. The accuracy of our algorithm compares favorably with that of the Givens rotation based algorithm while the Householder method may produce very sensitive results. Moreover, both fast and standard Givens rotation based algorithms produce very accurate results regardless of row sorting and even with extremely large weights, in our experiments. This makes the fast plane rotation a method of choice for the QR factorization since it is also competitive in complexity with the Householder method.

LSE problems arise in several applications, including:

- adaptive beamforming in signal processing [VL85],
- curve fitting, where a curve interpolates certain points or must match continuity conditions [Bjö90],
- penalty function methods in nonlinear optimization [VL83],
- geodetic least squares adjustment [VL85],
- voltera integral equations, and
- trust regions for optimization,
- also, in surface fitting, constraints such as positivity, monotonicity, and convexity are represented as inequality constraints which are solved for as sets of LSE's [LH74].

The methods developed to solve the LSE problem include the

- *nullspace method*,
- the *direct elimination method*,
- and the *weighting method* [LH74].

All of these methods involve orthogonalization at some stage. As methods based on the solution of normal equations perform much worse as problem matrix condition

numbers increase, they are not generally recommended. The LSE problem may also be solved via psuedo-inverses (and thus via the singular value decomposition (SVD)) [LH74], but because of its inefficiency, this method is of practical use for analysis only. More recent analyses use the perspective of the weighting method to yield a more complete analysis of the LSE problem. Van Loan [VL85] uses the generalized SVD (GSVD) in his analysis, and Eldén [Eld80, Eld82] bases a perturbation theory on weighted psuedo-inverses. Barlow [Bar88] offers a further error analysis of the LSE solver proposed by Van Loan [VL85], and shows that Van Loan's conjecture that the method converges for "all but ill-conditioned problems" is valid. Björck [Bjö90] explains that the nullspace and the direct elimination methods are numerically stable and cites Leringe and Wedin [LW70] who show that the two methods yield almost identical numerical accuracy results. There are also direct and iterative methods for the solution of sparse problems which are beyond the scope of this article. Readers are directed to [BNP88].

3.5.1 Weighted Rows and Equality Constraints

In this section, we briefly review weighting method, null space method, and the direct elimination method and compare their computational complexities. In the weighting method, the LSE problem (3.57) is transformed to the unconstrained linear least squares (LS) problem,

$$\min_x \left\| \begin{bmatrix} \eta B \\ A \end{bmatrix} x - \begin{bmatrix} \eta d \\ b \end{bmatrix} \right\|_2, \ \eta \gg 1, \tag{3.58}$$

which is the same as the weighted least squares problem (WLS) [Bjö90]

$$\min_x \left\| W \left(\begin{bmatrix} B \\ A \end{bmatrix} x - \begin{bmatrix} d \\ b \end{bmatrix} \right) \right\|_2, \tag{3.59}$$

where

$$W = diag(\eta I_l, \ I_m). \tag{3.60}$$

It is shown [GVL89] that with a large enough η, the solution to (3.58) can accurately approximate the LSE solution to (3.57). However, a large value of η yields a *stiff* problem and we have to choose the algorithm carefully for accurate results. The condition number, $\kappa(\cdot)$, of a matrix is adversely affected by its row weighting. However, the condition number based perturbation analyses of general least squares solutions will usually be much too pessimistic for the weighted case, since the perturbations in the weighted elements have a special structure, making the analysis of the general case inapplicable [Bjö90]. Still, the perturbations will be worse than for the unweighted case [Bjö90]. This expression can be considered as a special case of the generalized

linear least squares problem,

$$\min_x \left(\begin{bmatrix} B \\ A \end{bmatrix} x - \begin{bmatrix} d \\ b \end{bmatrix} \right) W^{-1} \left(\begin{bmatrix} B \\ A \end{bmatrix} x - \begin{bmatrix} d \\ b \end{bmatrix} \right), \; W \in \mathbb{R}^{(l+m)\times(l+m)}. \tag{3.61}$$

If W is symmetric and positive definite, it has the Cholesky factorization,

$$W^{-1} = G^T G. \tag{3.62}$$

where, in a statistical context, the regression coefficients, $g_{ij}, (i \neq j)$, reflect the coupling of the respective equations, and the g_{ii} elements represent the relative reliability of the respective observations. Normalized regression coefficients are correlation coefficients. The problem of finding the linear least squares solution to the weighted system of equations,

$$\min_x \|GAx - Gb\|_2 , \tag{3.63}$$

can be expressed in the normal equation form as

$$A^T W^{-1} A x = A^T W^{-1} b. \tag{3.64}$$

This equation is offered for theoretical purposes only, as computing from the normal equations explicitly can be computationally unstable. Weighted least squares can also represent a non-weighted minimization where a new length and inner product are defined:

$$\|GAx - Gb\|_2 = \|Ax - b\|_G, \tag{3.65}$$

where $\|Gy\|_2 = \|y\|_G$, $\forall y$, $\langle x, y \rangle_G = (Gx)^T (Gy)$, and $\langle x, x \rangle_G = \|x\|_G^2$.

With the restriction that the unscaled covariance matrix is diagonal,

$$W = \mathrm{diag}(w_1, \ldots, w_m), w_i > 0, \forall i. \tag{3.66}$$

The normal equation method of solution

$$(\eta^2 B^T B + A^T A)x = \eta^2 B^T d + A^T b \tag{3.67}$$

should be avoided when the $\eta^2 B^T B$ term overwhelms the $A^T A$ term resulting an unacceptable information loss for large values of η. except for certain notable instances such as when $B = \mathrm{diag}(D)$ where the $\eta^2 B^T B$ term has a benign effect.

The difference between the weighted and unweighted least squares solutions, x_W and x_{LS}, respectively, can be expressed as

$$x_W - x_{LS} = \left(C^T W^2 C\right)^{-1} C^T \left(W^2 - I\right)(f - C x_{LS}), \tag{3.68}$$

where

$$C \equiv \begin{bmatrix} B \\ A \end{bmatrix} \qquad \text{and} \qquad f \equiv \begin{bmatrix} d \\ b \end{bmatrix},$$

which shows the dependence of the solution on the weighting [GVL89]. The Householder, modified Gram-Schmidt, and Givens orthogonal factorization methods are commonly used for the solution of the LS problems [Bjö91, GVL89]. It has been shown that for the stiff problems, the solution vector obtained by the QR factorization [3] is sensitive to the row sorting of the matrix and also to the size of η. However, in this section, we will show that although this is the case with the Householder method, *the Givens method and its fast versions for the QR factorization are not sensitive to row sorting* according to a substantial number of experiments.

[3] As an important implementation detail, if matrices are stored in row-major form, the $X = QR$ factorization should be performed, else, if storage is in column-major form, the $X^T = LQ^T$ factorization should be performed so as to promote and maximize contiguous data reference. Herein we will assume row-major storage and discuss the QR rather than the LQ^T factorization.

The nullspace method [LH74], summarized in Algorithm 1, uses an orthogonal basis of the null space of the constraint matrix. Although this method admits stable updating, it is inefficient if A is large and sparse [GVL89] or if the problem is to be solved for various B matrices for a fixed A, as each product, AQ_B, must be recalculated.

Algorithm 1 (Nullspace)

$$\begin{aligned}
\begin{bmatrix} B \\ A \end{bmatrix} Q_B &\Longrightarrow \begin{bmatrix} L_B & 0 \\ \tilde{A}_1 & \tilde{A}_2 \end{bmatrix},\ Q_B^T Q_B = I,\ L_B \in \mathbb{R}^{l\times l}\ \textit{(LQ transform)} \\
L_B^{-1} d &\Longrightarrow \bar{x}_B\ \textit{(triangular forward solve)} \\
b - \tilde{A}_1 \bar{x}_B &\Longrightarrow \tilde{b} \\
\min_{\bar{x}_A} \left\| \tilde{A}_2 \bar{x}_A - \tilde{b} \right\|_2 &\Longrightarrow \bar{x}_A \in \mathbb{R}^{(n-l)}\ \textit{(LS solve)} \\
Q_B \begin{bmatrix} \bar{x}_B \\ \bar{x}_A \end{bmatrix} &\Longrightarrow x.
\end{aligned}$$

The direct elimination method is presented in Algorithm 2. It applies orthogonal transformations to the constraint matrix and then elementary transformations (Gaussian elimination) to the data matrix. Column interchanges are necessary to insure that the resulting first l columns of the constraint matrix are linearly independent. Then, the equality constrained least squares problem (3.57), can be restated as the unconstrained problem. For the generalization of this algorithm to handle the instance where A and B have certain sparsity structure and each may be rank deficient, see [Bjö84]. The direct elimination method admits updating in a straightforward manner.

Table 3.1: **Computational Complexity of the LSE Solvers**

weighting	null space	direct elimination
$\frac{1}{2}\nu n^2\left(m+l-\frac{n}{3}\right)$	$\frac{1}{2}\nu n^2\left(m+l-\frac{n}{3}\right)$	$\frac{1}{2}\nu n^2\left(m+l-\frac{n}{3}\right)-mnl\,(\nu-2)$

ν = operations for a 2 element rotation. $\nu = 6$ flops (slow rotation), $\nu = 4$ flops (fast rotation)

Algorithm 2 (Direct Elimination)

$$\begin{bmatrix} Q_B^T & 0 \\ 0 & I_m \end{bmatrix}\begin{bmatrix} B & d \\ A & b \end{bmatrix}\Pi \implies Q_B^T\begin{bmatrix} B_1 & B_2 & d \\ A_1 & A_2 & b \end{bmatrix} = \begin{bmatrix} R_B & \tilde{B}_2 & \tilde{d} \\ A_1 & A_2 & b \end{bmatrix},$$

$B_1 \in \mathbb{R}^{l\times l}, R_B \in \mathbb{R}^{l\times l}$ *(QR with column interchange)*

$$\begin{bmatrix} I_l & 0 \\ M & I_m \end{bmatrix}\begin{bmatrix} R_B & \tilde{B}_2 & \tilde{d} \\ A_1 & A_2 & b \end{bmatrix} \implies \begin{bmatrix} R_B & \tilde{B}_2 & \tilde{d} \\ 0 & \tilde{A}_2 & \tilde{b} \end{bmatrix} \textit{ (Gaussian elim.)}$$

$$\begin{bmatrix} I_l & 0 \\ 0 & Q_A^T \end{bmatrix}\begin{bmatrix} R_B & \tilde{B}_2 & \tilde{d} \\ 0 & \tilde{A}_2 & \tilde{b} \end{bmatrix} \implies \begin{bmatrix} R_B & \tilde{B}_2 & \tilde{d} \\ 0 & R_A & \hat{b}_1 \\ 0 & 0 & \hat{b}_2 \end{bmatrix},\ R_A \in \mathbb{R}^{(n-l)\times(n-l)}$$

$$\begin{bmatrix} R_B & \tilde{B}_2 \\ 0 & R_A \end{bmatrix}^{-1}\begin{bmatrix} \tilde{d} \\ \hat{b}_1 \end{bmatrix} \implies \bar{x} \textit{ (triangular backsolve)}$$

$$\Pi\bar{x} \implies x.$$

In Table 1, the floating point complexity (1 flop $\approx$ 1 multiplication or 1 addition) is presented using the efficient algorithm for each method. It was assumed that the orthogonal factor was not formed explicitly but the rotations which would form it were saved in factored form and later used as described in [Ste76].

Because the null space and weighting methods consist of rotations and triangular

solves exclusively, their floating point complexities are roughly the same. The direct elimination method replaces some orthogonal transformations with elementary transformations which are computationally less expensive. However, this may result in less accurate results, e.g. when the elements in A are much larger than those in B.

3.6 Column Pivoting and Multiple Constraints

In the previous section, we have shown that fast rotations are not sensitive to row sorting. This is not the case when Householder transformation is used for the QR factorization. The QR factorizations by Householder reflections are stable for stiff problems if the proper row interchanges are effected [PR69, Bjö90]. However, A 4×3 exemplary matrix in [PR69] demonstrates the poor accuracy which resulted from an improper row sorting when a Householder QR factorization is performed. In [VL85], there are examples that illustrate that B-over-A as shown in (3.59) produce much more accurate solutions than A-over-B when Householder QR factorization is performed.

Another example in [VL85] shows that column pivoting is also necessary for improved accuracy in the solution. The following theorem due to Stewart [Ste84] illuminates the importance of column pivoting in the weighting method.

Theorem 1 *Let the weighted augmented matrix be*

$$\begin{bmatrix} B \\ A \end{bmatrix} = \begin{bmatrix} B_1 & B_2 \\ \epsilon A_1 & \epsilon A_2 \end{bmatrix}, \; B_1 \in \mathbb{R}^{l \times l}, \tag{3.69}$$

and let R_{11} be the triangular factor of the QR factorization of B_1. Then the QR factorization of $\begin{bmatrix} B \\ A \end{bmatrix}$ *is*

$$\begin{bmatrix} B_1 & B_2 \\ \epsilon A_1 & \epsilon A_2 \end{bmatrix} = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_{11} + \mathcal{O}(\epsilon^2) & R_{11}^{-1} B_2 + \mathcal{O}(\epsilon^2) \\ 0 & \epsilon \bar{R}_{22} + \mathcal{O}(\epsilon^3) \end{bmatrix}, \tag{3.70}$$

where $\bar{A}_2 = \bar{Q}_{22}\bar{R}_{22}$ *is the QR factorization of* $\bar{A}_2 = A_2 - A_1 B_1^{-1} B_2$, *with*

$$Q_1 = \begin{bmatrix} B_1 R_{11}^{-1} + \mathcal{O}(\epsilon^2) \\ \epsilon A_1 R_{11}^{-1} + \mathcal{O}(\epsilon^3) \end{bmatrix} \quad \textit{and } Q_2 = \begin{bmatrix} -\epsilon B_1^{-T} A_1^T + \mathcal{O}(\epsilon^3) \\ I + \mathcal{O}(\epsilon^2) \end{bmatrix} \bar{Q}_{22}. \tag{3.71}$$

The keystone of this theorem is that the submatrix B_1 must be well conditioned for a stable algorithm, which corroborates the proof of Powell and Reid in [PR69] that column pivoting should be used in solving the WLS problem. This also shows that the solution by the weighting method is an approximation to the solution by the direct elimination method. Therefore, column pivoting is also important in the weighting method. In fact, when $\eta \to \infty$ in the weighting method, the direct elimination method is obtained. Note that the Givens rotations applied to weighted and unweighted row pairs in

$$\frac{1}{\sqrt{\eta^2 a_{pi}^2 + a_{qi}^2}} \begin{bmatrix} \eta a_{pi} & a_{qi} \\ -a_{qi} & \eta a_{pi} \end{bmatrix} \begin{bmatrix} \eta a_{pi} \\ a_{qi} \end{bmatrix} = \begin{bmatrix} \eta a'_{pi} \\ 0 \end{bmatrix} \tag{3.72}$$

becomes

$$\begin{bmatrix} 1 & 0 \\ -a_{qi}/a_{pi} & 1 \end{bmatrix} \begin{bmatrix} a_{pi} \\ a_{qi} \end{bmatrix} = \begin{bmatrix} a'_{pi} \\ 0 \end{bmatrix} \tag{3.73}$$

after a row and column scaling as $\eta \to \infty$, which is a Gaussian elimination step in the direct elimination method.

Applying column pivoting to the weighting method essentially does not change the computational complexity. However, in many applications, such as adaptive beamforming, the LSE needs to be solved for many different constraint matrices B for each fixed data matrix A. As mentioned in [VL85], the A matrix can be first orthogonally triangularized as $A = QR$. The computational cost of this operation

will be, for $m \geq n$, $\frac{1}{2}\nu n^2\left(m - \frac{n}{3}\right)$ flops. [4] Then for each constraint matrix B, we can triangularize the matrix

$$C = \begin{pmatrix} \eta B \\ R \end{pmatrix}. \tag{3.74}$$

For $m \gg n$, the precomputation of the QR factorization of A will result in significant savings for the direct elimination and weighting methods. However, the orthogonal transformations from the right in the null space method may lead to a complete fill in of the empty half of the triangular matrix. Thus, the floating point complexity of the null space method for each new set of equality constraints will be $\mathcal{O}(n^3)$, which is significantly more work than what is required by the other two methods.

We can triangularize (3.74) by first applying the orthogonal transformations to B matrix to make it upper trapezoidal, in $\frac{1}{2}\nu l^2\left(n - \frac{l}{3}\right)$ flops. Then the direct elimination and the weighting methods have $\mathcal{O}(ln)$ elements to annihilate at a expense of $\frac{1}{2}\nu l\left(n(n-l) + \frac{l^2}{3}\right)$ flops. The direct elimination method is a little less expensive because $\frac{l^2}{2}$ elements can be eliminated via elementary transformations.

We now discuss how the column pivoting can be efficiently incorporated in the weighting method for solving the multiple constraint problem. We first obtain the QR factorization of the A matrix. In the next process of triangularizing the matrix C, column pivoting may significantly increase the computational complexity since it can destroy the triangular structure of R. However, the constraint matrix B typically has a very small number of rows such as in beamforming. Since we want to have well conditioned matrix B_1, we can apply column pivoting only in the first l steps of triangularizing the matrix C. The orthogonal transformations necessary to triangularize C with this pivoting will introduce at most ln extra nonzero elements to annihilate.

[4] ν represents the number of flops used to rotate a single pair of elements ($\nu = 6$–slow, $\nu = 4$–fast)

Thus, we can achieve better stability without increasing the complexity significantly.

The standard pivoting algorithm chooses as the pivot column the column having the largest Euclidean norm in the partial factorization. An alternative modification of this algorithm they label *threshold* pivoting, where the pivot column is the closest column whose Euclidean norm is at least as large as some prespecified proper fraction of the column with the largest Euclidean norm. This algorithm endeavors to preserve the existing ordering as much as possible while still selecting reasonably independent pivot columns. Another alternative is to bound the distance between exchange indices for either the standard or the threshold pivoting algorithm. The extreme case of bounding the exchange index distance is pairwise pivoting which affords much of the safety of the standard pivoting algorithm at the minimum computational expense.

Björck [Bjö90] cites results of Stewart which show that column pivoting, analogous to partial pivoting in Gaussian elimination, performs very well in practice. Where the $\kappa(A)$ is underestimated by at most a order of magnitude by $|r_{11}/r_{nn}|$. However, pathological examples have been shown. Because the A matrix is factored only once, and the B matrix has so few rows, a rank-revealing QR factorization (RRQR) should be considered. Bischof and Hansen [BCH91] propose a RRQR factorization well suited to the constraint update problem. Their algorithm uses a restricted column pivoting strategy in conjunction with incremental condition estimation during the initial QR factorization followed by a Chan/Foster backward pass which improves or validates the factorization to be a RRQR factorization. Typically, because of the restricted pivoting in the initial factorization, little extra work is required during the backward pass. The restricted pivoting algorithm may be pairwise which yields the least fill in and subsequent retriangularization work.

Chapter IV

ERROR ANALYSIS AND NUMERICAL RESULTS

4.1 Error Analysis

4.1.1 Standard Fast Rotations

We first review the error analysis of the standard fast rotation for the QR factorization which is based upon Parlett [Par80]. A rotation of two rows, p and q, may be represented as

$$\begin{bmatrix} \tilde{d}_p & 0 \\ 0 & \tilde{d}_q \end{bmatrix} \begin{bmatrix} \tilde{y}_{p1} & \tilde{y}_{p2} & \cdots & \tilde{y}_{pn} \\ 0 & \tilde{y}_{q2} & \cdots & \tilde{y}_{qn} \end{bmatrix} = \begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} d_p & 0 \\ 0 & d_q \end{bmatrix} \begin{bmatrix} y_{p1} & y_{p2} & \cdots & y_{pn} \\ y_{q1} & y_{q2} & \cdots & y_{qn} \end{bmatrix},$$

where

$$\begin{bmatrix} \tilde{y}_{p1} & \tilde{y}_{p2} & \cdots & \tilde{y}_{pn} \\ 0 & \tilde{y}_{q2} & \cdots & \tilde{y}_{qn} \end{bmatrix} = \begin{bmatrix} 1 & -\alpha \\ \beta & 1 \end{bmatrix} \begin{bmatrix} y_{p1} & y_{p2} & \cdots & y_{pn} \\ y_{q1} & y_{q2} & \cdots & y_{qn} \end{bmatrix}.$$

Also, let $\tau \equiv \tan\theta$ and $\omega \equiv \cos^2\theta$.

Let ϵ, $|\epsilon| \ll 1$, which may be different at every instance, be a tiny number close to *machine*-ϵ. The inclusion of floating point roundoff error in a computed quantity will be represented by the product of the exact theoretical value with $(1+\epsilon)$, with the notational shorthand of representing $(1+\epsilon_1)(1+\epsilon_2)$ as $(1+\epsilon)^2$. We will assume that the initial variables are exact, and computed quantities will be denoted by primes.

The computed scalars are

$$
\begin{aligned}
\alpha' &= \alpha(1+\epsilon) && \{= y_{q1}/y_{p1}\} \\
\beta' &= \beta(1+\epsilon)^3 && \left\{= \alpha' d_q^2/d_p^2\right\} \\
\omega' &= \omega(1+\epsilon)^6 && \left\{= 1/\left(1+\left(\tfrac{d_q y_q}{d_p y_p}\right)^2\right)\right\} \\
&\approx (c(1+\epsilon)^3)^2
\end{aligned} \tag{4.1}
$$

Although we have only the squared values of d_p and d_q in actual computation, we will use d_p and d_q in this analysis. Then, after the rotation,

$$
\left\{ \begin{array}{rcl} \tilde{d}'_p &=& d_p c(1+\epsilon)^4 \\ \tilde{d}'_q &=& d_q c(1+\epsilon)^4 \end{array} \right\}, \tag{4.2}
$$

and the vector equations are

$$
\left\{ \begin{array}{rcl} \tilde{y}'_{pi} &=& \left[y_{pi} + \beta' y_{qi}(1+\epsilon)\right](1+\epsilon) = y_{pi}(1+\epsilon) + \beta y_{qi}(1+\epsilon)^5 \\ \tilde{y}'_{qi} &=& \left[y_{qi} - \alpha' y_{pi}(1+\epsilon)\right](1+\epsilon) = y_{qi}(1+\epsilon) - \alpha y_{pi}(1+\epsilon)^3 \end{array} \right\}. \tag{4.3}
$$

¿From Equations (4.2) and (4.3),

$$
\left\{ \begin{array}{rcl} \tilde{d}'_p\, \tilde{y}'_{pi} &=& d_p c y_{pi}(1+\epsilon)^5 + d_p c\beta y_{qi}(1+\epsilon)^9 \\ \tilde{d}'_q\, \tilde{y}'_{qi} &=& d_q c y_{qi}(1+\epsilon)^5 - d_q c\alpha y_{pi}(1+\epsilon)^7 \end{array} \right\}, \tag{4.4}
$$

and using the first order approximation, $1 + k\epsilon \approx (1+\epsilon)^k$, for any integer k,

$$
\left\{ \begin{array}{rcl} \tilde{d}'_p\, \tilde{y}'_{pi} &=& d_p c y_{pi}(1+5\epsilon) + d_q s y_{qi}(1+9\epsilon) \\ \tilde{d}'_q\, \tilde{y}'_{qi} &=& d_q c y_{qi}(1+5\epsilon) - d_p s y_{pi}(1+7\epsilon) \end{array} \right\} \tag{4.5}
$$

Extracting the error terms, we have the equation,

$$
\left[\begin{array}{c} \tilde{d}'_p\, \tilde{y}'_{pi} \\ \tilde{d}'_q\, \tilde{y}'_{qi} \end{array} \right] = \left[\begin{array}{c} \tilde{d}_p \tilde{y}_{pi} \\ \tilde{d}_q \tilde{y}_{qi} \end{array} \right] + \left[\begin{array}{cc} 5c\epsilon & 9s\epsilon \\ -7s\epsilon & 5c\epsilon \end{array} \right] \left[\begin{array}{c} d_p y_{pi} \\ d_q y_{qi} \end{array} \right], \tag{4.6}
$$

which is of exactly the same form as the equation for the errors incurred in a standard Givens rotation [Gen73a]. The same analysis and conclusion holds for the large angle formulae. Eqn.(4.26) may be reformulated as the inequality,

$$\left|\begin{bmatrix} \tilde{d}'_p\,\tilde{y}'_{pi} \\ \tilde{d}'_q\,\tilde{y}'_{qi} \end{bmatrix} - \begin{bmatrix} \tilde{d}_p\tilde{y}_{pi} \\ \tilde{d}_q\tilde{y}_{qi} \end{bmatrix}\right| \leq \left|\begin{bmatrix} 5c\epsilon & 9s\epsilon \\ 7s\epsilon & 5c\epsilon \end{bmatrix}\right| \cdot \left|\begin{bmatrix} d_p y_{pi} \\ d_q y_{qi} \end{bmatrix}\right|, \tag{4.7}$$

where $|A| = (|a_{ij}|)$.

4.1.2 New Fast Rotations

To differentiate the computed scalars in the new fast rotations from those of the standard fast rotations, we will use the *hat* notation, '^', whenever it is necessary. The computed scalars in the new fast rotations are the same as those in Eqn. (4.1) except that α is redefined as $\hat{\alpha}$:

$$\hat{\alpha}' \;=\; \hat{\alpha}(1+\epsilon)^8 \quad \{= \alpha'\omega' = \alpha\omega(1+\epsilon)^8\} \tag{4.8}$$

As before, the unsquared values of d_p and d_q will be used. However, the change from d_q to $\hat{d}_q$ should be noted:

$$\left\{\begin{array}{rcl} \tilde{d}'_p & = & d_p c(1+\epsilon)^4 \\ \hat{d}_q{}' & = & (d_q/c)(1+\epsilon)^4 \end{array}\right\} \tag{4.9}$$

The chained vector equations are:

$$\left\{\begin{array}{rcl} \tilde{y}'_{pi} & = & \left[y_{pi} + \beta' y_{qi}(1+\epsilon)\right](1+\epsilon) \\ \hat{y}'_{qi} & = & \left[y_{qi} - \hat{\alpha}'\,\tilde{y}'_{pi}(1+\epsilon)\right](1+\epsilon) \end{array}\right\} \tag{4.10}$$

Expanding (4.10), we get

$$(4.11)\qquad \left\{ \begin{array}{rcl} \tilde{y}'_{pi} & = & y_{pi}(1+\epsilon) + \beta y_{qi}(1+\epsilon)^5 \\ \hat{y}'_{qi} & = & y_{qi}(1+\epsilon) - \alpha\omega\left[y_{pi}(1+\epsilon) + \beta y_{qi}(1+\epsilon)^5\right](1+\epsilon)^{10} \\ & = & \left[1 - \alpha\omega\beta(1+\epsilon)^{14}\right] y_{qi}(1+\epsilon) - \alpha\omega y_{pi}(1+\epsilon)^{11} \end{array} \right\}$$

Factoring in the diagonal elements by merging (4.9) and (4.11) with approximation of ϵ terms up to the first order, we have (noting that $(1-\alpha\omega\beta(1+\epsilon)^{14})(1+\epsilon) \approx (c^2(1+\epsilon) - 14s^2\epsilon)$),

$$(4.12)\qquad \left\{ \begin{array}{rcl} \tilde{d}'_p \tilde{y}'_{pi} & = & d_p c y_{pi}(1+5\epsilon) + d_q s y_{qi}(1+9\epsilon) \\ \hat{d}_q{}' \hat{y}'_{qi} & = & d_q y_{qi}\left[c(1+5\epsilon) - 14s\tau\epsilon\right] - d_p s y_{pi}(1+15\epsilon) \end{array} \right\}$$

Separating out the error terms,

$$(4.13)\qquad \left[\begin{array}{c} \tilde{d}'_p \tilde{y}'_{pi} \\ \hat{d}_q{}' \hat{y}'_{qi} \end{array} \right] = \left[\begin{array}{c} \tilde{d}_p \tilde{y}_{pi} \\ \hat{d}_q \hat{y}_{qi} \end{array} \right] + \left[\begin{array}{cc} 5c\epsilon & 9s\epsilon \\ -15s\epsilon & (5c - 14s\tau)\epsilon \end{array} \right] \left[\begin{array}{c} d_p y_{pi} \\ d_q y_{qi} \end{array} \right].$$

Since $|\theta| \le \pi/4$, $|5c - 14s\tau| \le |9c|$, and the following inequality may be formulated

$$(4.14)\qquad \left\| \left[\begin{array}{c} \tilde{d}'_p \tilde{y}'_{pi} \\ \hat{d}_q{}' \hat{y}'_{qi} \end{array} \right] - \left[\begin{array}{c} \tilde{d}_p \tilde{y}_{pi} \\ \hat{d}_q \hat{y}_{qi} \end{array} \right] \right\| \le \left\| \left[\begin{array}{cc} 5c\epsilon & 9s\epsilon \\ 15s\epsilon & 9c\epsilon \end{array} \right] \right\| \cdot \left\| \left[\begin{array}{c} d_p y_{pi} \\ d_q y_{qi} \end{array} \right] \right\|$$

The error analysis for the fast rotation when the q-th row is updated first is procedurally identical and yields proportional results:

$$(4.15)\qquad \hat{\beta}' = \hat{\beta}(1+\epsilon)^{10} \quad \{= \beta'\omega' = \beta\omega(1+\epsilon)^{10}\}$$

$$(4.16)\qquad \left\{ \begin{array}{rcl} \tilde{d}'_q & = & d_q c(1+\epsilon)^4 \\ \hat{d}_p{}' & = & (d_p/c)(1+\epsilon)^4 \end{array} \right\}$$

$$(4.17)\qquad \left\{ \begin{array}{rcl} \tilde{y}'_{qi} & = & \left[y_{qi} - \alpha' y_{pi}(1+\epsilon)\right](1+\epsilon) \\ \hat{y}'_{pi} & = & \left[y_{pi} - \hat{\alpha}' \tilde{y}'_{qi}(1+\epsilon)\right](1+\epsilon) \end{array} \right\}$$

$$(4.18)\qquad \left\{ \begin{array}{rcl} \tilde{y}'_{qi} &=& y_{qi}(1+\epsilon) - \alpha y_{pi}(1+\epsilon)^3 \\ \hat{y}'_{pi} &=& y_{pi}(1+\epsilon) + \beta\omega\left[y_{qi}(1+\epsilon) - \alpha y_{pi}(1+\epsilon)^3\right](1+\epsilon)^{12} \\ &=& \left[1 - \alpha\omega\beta(1+\epsilon)^{14}\right] y_{pi}(1+\epsilon) + \beta\omega y_{qi}(1+\epsilon)^{13} \end{array} \right\}$$

$$(4.19)\qquad \left\{ \begin{array}{rcl} \tilde{d}'_q \tilde{y}'_{qi} &=& d_q c y_{qi}(1+5\epsilon) - d_p s y_{pi}(1+7\epsilon) \\ \hat{d}_p{}' \hat{y}'_{pi} &=& d_p y_{pi}\left[c(1+5\epsilon) - 14 s\tau\epsilon\right] + d_q s y_{pi}(1+17\epsilon) \end{array} \right\}$$

$$(4.20)\qquad \left[\begin{array}{c} \tilde{d}'_q\, \tilde{y}'_{qi} \\ \hat{d}_p{}'\, \hat{y}'_{pi} \end{array} \right] = \left[\begin{array}{c} \tilde{d}_q \tilde{y}_{qi} \\ \hat{d}_p \hat{y}_{pi} \end{array} \right] + \left[\begin{array}{cc} 5c\epsilon & -7s\epsilon \\ 17s\epsilon & (5c - 14s\tau)\epsilon \end{array} \right] \left[\begin{array}{c} d_q y_{qi} \\ d_p y_{pi} \end{array} \right]$$

Finally, Eqn. (4.20) yields the inequality,

$$(4.21)\qquad \left\| \left[\begin{array}{c} \tilde{d}'_q\, \tilde{y}'_{qi} \\ \hat{d}_p{}'\, \hat{y}'_{pi} \end{array} \right] - \left[\begin{array}{c} \tilde{d}_q \tilde{y}_{qi} \\ \hat{d}_p \hat{y}_{pi} \end{array} \right] \right\| \leq \left\| \left[\begin{array}{cc} 5c\epsilon & 7s\epsilon \\ 17s\epsilon & 9c\epsilon \end{array} \right] \right\| \cdot \left\| \left[\begin{array}{c} d_q y_{qi} \\ d_p y_{pi} \end{array} \right] \right\|$$

As the derivations (for the error analyses of the large angle formulae) are similar to those of the small angle formulae, we will state only the results. For $|\theta| > \pi/4$ and when the q-th row is updated based on the updated p-th row ($p < q$),

$$(4.22)\qquad \left[\begin{array}{c} \tilde{d}'_p\, \tilde{y}'_{pi} \\ \hat{d}_q{}'\, \hat{y}'_{qi} \end{array} \right] = \left[\begin{array}{c} \tilde{d}_p \tilde{y}_{pi} \\ \hat{d}_q \hat{y}_{qi} \end{array} \right] + \left[\begin{array}{cc} 9c\epsilon & 5s\epsilon \\ -(5s - 14c\tau^{-1})\epsilon & 15c\epsilon \end{array} \right] \left[\begin{array}{c} d_p y_{pi} \\ d_q y_{qi} \end{array} \right]$$

$$(4.23)\qquad \left\| \left[\begin{array}{c} \tilde{d}'_p\, \tilde{y}'_{pi} \\ \hat{d}_q{}'\, \hat{y}'_{qi} \end{array} \right] - \left[\begin{array}{c} \tilde{d}_p \tilde{y}_{pi} \\ \hat{d}_q \hat{y}_{qi} \end{array} \right] \right\| \leq \left\| \left[\begin{array}{cc} 9c\epsilon & 5s\epsilon \\ 9s\epsilon & 15c\epsilon \end{array} \right] \right\| \cdot \left\| \left[\begin{array}{c} d_p y_{pi} \\ d_q y_{qi} \end{array} \right] \right\|$$

and, for the counter-chained rotation (the p-th row is updated based on the updated q-th row, $p < q$),

$$(4.24)\qquad \left[\begin{array}{c} \tilde{d}'_q\, \tilde{y}'_{qi} \\ \hat{d}_p{}'\, \hat{y}'_{pi} \end{array} \right] = \left[\begin{array}{c} \tilde{d}_q \tilde{y}_{qi} \\ \hat{d}_p \hat{y}_{pi} \end{array} \right] + \left[\begin{array}{cc} 7c\epsilon & -5s\epsilon \\ (5s - 14c\tau^{-1})\epsilon & 17c\epsilon \end{array} \right] \left[\begin{array}{c} d_q y_{qi} \\ d_p y_{pi} \end{array} \right]$$

$$(4.25) \qquad \left\| \begin{bmatrix} \tilde{d}'_q \, \tilde{y}'_{qi} \\ \hat{d}_p{}' \, \hat{y}'_{pi} \end{bmatrix} - \begin{bmatrix} \tilde{d}_q \tilde{y}_{qi} \\ \hat{d}_p \hat{y}_{pi} \end{bmatrix} \right\| \leq \left\| \begin{bmatrix} 7c\epsilon & 5s\epsilon \\ 9s\epsilon & 17c\epsilon \end{bmatrix} \right\| \cdot \left\| \begin{bmatrix} d_q y_{qi} \\ d_p y_{pi} \end{bmatrix} \right\|$$

As we can see from Equations (4.14), (4.21), (4.23), and (4.25), because of the chaining between the two linked triads, there is the potential for roughly doubled error in the vector which is updated second, compared to the standard fast rotations. However, the rotation is still stable, as the magnitudes of the elements of the error matrix are bounded by a small constant. For stability, it is important that the small angle formulae must be used when $|\tau| \leq 1$, and the large angle formulae are required otherwise. Error analyses of other algorithms which employ fast rotations, e.g. the Jacobi and Hestenes algorithms, would proceed similarly, and would yield similar stability results.

4.1.3 Row Pivoting in Self-Scaling Fast Rotations

In this section, we present a reformulation of the error analysis for the self-scaling square-root free fast rotations and discuss the role of row sorting in producing accurate results. The following analysis is based on the error analysis of the standard fast Givens rotation as derived by Parlett [Par80]. We assume that we are at some intermediate stage in the QR factorization by self-scaling fast rotations. Let ϵ, which may be different at every instance, be a tiny number such that $|\epsilon| \leq \mathbf{u}$ where $\mathbf{u}$ is the *unit roundoff* of the architecture, and let it represent the standard error of an elementary floating-point operation. We will use the notational shorthand of representing $\prod_1^k(1+\epsilon)$ as $(1+k\epsilon)$ and a computed quantity will be differentiated from its exact value by the following convention: $\mathrm{fl}(\alpha) = \hat{\alpha}$. An updated quantity will be represented with a *prime* affixed.

We now examine the small angle case $|\theta| \leq \frac{\pi}{4}$, with $|d_p| \geq |d_q|$, and the large angle case $\frac{\pi}{2} > |\theta| > \frac{\pi}{4}$, with $|d_p| < |d_q|$. The small angle case arises when $|x_{p1}| > |x_{q1}|$. This will usually be the case when x_{p1} is an element of the constraint matrix B but x_{q1} is not, in the weighting method since the constraint is heavily weighted.

The scalars computed in floating point arithmetic are shown in Table 3 in the same order they appear in Table 2. As an improvement over previous analyses [AP94a], we introduce the new error variable ϵ_δ to reflect the different nature of that operation's error behavior and to yield a tighter error bound. Note that for $|\theta| \leq \frac{\pi}{4}$, if $|\theta|$ diminishes, $\tau\beta = t^2$ correspondingly diminishes, and thus the more least-significant bits will be shifted away in the process of normalization when added to 1. The possible error in $\tau\beta$ will be exhibited in the least-significant bits, so smaller angles correspond to equal or smaller maximum errors in δ, $|\theta_1| < |\theta_2| \Rightarrow \max|\hat{\delta}(\theta_1) -$

Table 4.1: **Self-Scaling Fast Rotation Scalars Floating Point Intermediate Errors**

$\lvert\theta\rvert \le \frac{\pi}{4},\ \lvert d_p\rvert \ge \lvert d_q\rvert$	$\frac{\pi}{2} > \lvert\theta\rvert > \frac{\pi}{4},\ \lvert d_p\rvert < \lvert d_q\rvert$
$\hat{\gamma} = \frac{d_p^2}{d_q^2}(1+\epsilon)$	$\hat{\gamma} = \frac{d_p^2}{d_q^2}(1+\epsilon)$
$\hat{\tau} = \frac{y_{q1}}{y_{p1}}(1+\epsilon) \quad \left\{= t\frac{d_p}{d_q}\right\}$	$\hat{\tau} = \frac{y_{p1}}{y_{q1}}(1+\epsilon) \quad \{= (t\gamma)^{-1}\}$
$\hat{\beta} = \frac{\tau}{\gamma}(1+3\epsilon) \quad \left\{= \frac{d_q^2}{d_p^2}\frac{y_{q1}}{y_{p1}} = t\frac{d_q}{d_p}\right\}$	$\hat{\beta} = \tau\gamma(1+3\epsilon) \quad \{= \frac{\gamma}{t}\}$
$\hat{\delta} = (1+\tau\beta(1+5\epsilon))(1+\epsilon) \quad \{= c^{-2}\}$	$\hat{\delta} = (1+\tau\beta(1+5\epsilon))(1+\epsilon) \quad \{= s^{-2}\}$
$= (1+\tau\beta)(1+\epsilon_\delta) \quad \{1 \ge \delta \ge 2\}$	$= (1+\tau\beta)(1+\epsilon_\delta) \quad \{1 \ge \delta \ge 2\}$
$\hat{\alpha} = \frac{\tau}{\delta}(1+2\epsilon+\epsilon_\delta) \quad \left\{= sc\frac{d_p}{d_q}\right\}$	$\hat{\alpha} = \frac{\tau}{\delta}(1+2\epsilon+\epsilon_\delta) \quad \left\{= sc\frac{d_q}{d_p} = sc/\gamma\right\}$
$\hat{r} = y_{p1}\delta(1+\epsilon+\epsilon_\delta)$	$\hat{r} = y_{q1}\delta(1+\epsilon+\epsilon_\delta)$
$\hat{d_p^2}' = \frac{d_p^2}{\delta}(1+\epsilon+\epsilon_\delta) \quad \left\{= c^2 d_p^2\right\}$	$\hat{d_p^2}' = \frac{d_q^2}{\delta}(1+\epsilon+\epsilon_\delta) \quad \{= s^2 d_q\}$
$\hat{d_q^2}' = d_p^2\delta(1+\epsilon+\epsilon_\delta) \quad \left\{= \frac{d_p^2}{c^2}\right\}$	$\hat{d_q^2}' = d_p^2\delta(1+\epsilon+\epsilon_\delta) \quad \left\{= \frac{d_p^2}{s^2}\right\}$

$\delta(\theta_1)| \le \max|\hat{\delta}(\theta_2) - \delta(\theta_2)|$. This relation may not hold if $|\theta_1|$ is too close to $|\theta_2|$. It should be noted though that even if $|(\hat{\tau}\hat{\beta} - \tau\beta)/\tau\beta| \approx \mathbf{u}$, and $|(\hat{\delta} - \delta)/\delta| \approx \mathbf{u}$, it is possible, for $|\theta| \ll \frac{\pi}{4}$, that $|((\hat{\delta} - 1) - \tau\beta)/\tau\beta| \gg \mathbf{u}$. This is indicative of unavoidable information loss in the calculation. For sufficiently small angles, $|(\hat{\delta}(\hat{\alpha}\hat{\beta}) - \delta(\alpha\beta))/\delta(\alpha\beta)| \le \mathbf{u}$, which insures high accuracy for $\hat{r}$, $\hat{d_p^2}'$ and $\hat{d_q^2}'$. An analogous formulation holds for the $\frac{\pi}{2} > |\theta| > \frac{\pi}{4}$ case.

In Table 4, the computed constants and associated errors, followed by the merging of the diagonal weight matrix are shown. In the small angle case, the ϵ_c term is introduced to facilitate the trigonometric substitution in the analysis. It also shows that the algebraic representation of a small number subtracted from 1 is indirectly

similar to the ϵ_δ term since ϵ_c tends to diminish for progressively smaller angles. Correspondingly, in the large angle case, the ϵ_s term, analogous to the ϵ_c term of the small angle analysis, will behave similar to the ϵ_δ term by tending to diminish for progressively larger angles.

These equations correspond closely to Parlett's results for the standard slow and standard fast rotations. Extracting the error terms for the small and large angle rotations, respectively, we have the equations,

$$(4.26)\quad \begin{bmatrix} \hat{d_p}'\hat{y_p}' \\ \hat{d_q}'\hat{y_q}' \end{bmatrix} = \begin{bmatrix} d'_p y'_p \\ d'_q y'_q \end{bmatrix} + \begin{bmatrix} (2\epsilon+\epsilon_\delta)c & (6\epsilon+\epsilon_\delta)s \\ -(6\epsilon+2\epsilon_\delta)s & (2\epsilon+\epsilon_\delta+\epsilon_c)c \end{bmatrix} \begin{bmatrix} d_p y_p \\ d_q y_q \end{bmatrix},$$

$$(4.27)\quad \begin{bmatrix} \hat{d_p}'\hat{y_p}' \\ \hat{d_q}'\hat{y_q}' \end{bmatrix} = \begin{bmatrix} d'_p y'_p \\ d'_q y'_q \end{bmatrix} + \begin{bmatrix} (2\epsilon+\epsilon_\delta)s & (6\epsilon+\epsilon_\delta)c \\ (6\epsilon+2\epsilon_\delta)c & -(2\epsilon+\epsilon_\delta+\epsilon_s)s \end{bmatrix} \begin{bmatrix} d_q y_q \\ d_p y_p \end{bmatrix},$$

which may be represented respectively as the following two inequalities,

$$(4.28)\quad \left\| \begin{bmatrix} \hat{d_p}'\hat{y_p}' \\ \hat{d_q}'\hat{y_q}' \end{bmatrix} - \begin{bmatrix} d'_p y'_p \\ d'_q y'_q \end{bmatrix} \right\| \leq \left\| \begin{bmatrix} (2\epsilon+\epsilon_\delta)c & (6\epsilon+\epsilon_\delta)s \\ -(6\epsilon+2\epsilon_\delta)s & (2\epsilon+\epsilon_\delta+\epsilon_c)c \end{bmatrix} \right\| \cdot \left\| \begin{bmatrix} d_p y_p \\ d_q y_q \end{bmatrix} \right\|,$$

$$(4.29)\quad \left\| \begin{bmatrix} \hat{d_p}'\hat{y_p}' \\ \hat{d_q}'\hat{y_q}' \end{bmatrix} - \begin{bmatrix} d'_p y'_p \\ d'_q y'_q \end{bmatrix} \right\| \leq \left\| \begin{bmatrix} (2\epsilon+\epsilon_\delta)s & (6\epsilon+\epsilon_\delta)c \\ (6\epsilon+2\epsilon_\delta)c & -(2\epsilon+\epsilon_\delta+\epsilon_s)s \end{bmatrix} \right\| \cdot \left\| \begin{bmatrix} d_q y_q \\ d_p y_p \end{bmatrix} \right\|.$$

An analogous analysis of the two alternate small and large angle formulae, respectively, yields the similar inequalities:

$$(4.30)\quad \left\| \begin{bmatrix} \hat{d_p}'\hat{y_p}' \\ \hat{d_q}'\hat{y_q}' \end{bmatrix} - \begin{bmatrix} d'_p y'_p \\ d'_q y'_q \end{bmatrix} \right\| \leq \left\| \begin{bmatrix} (2\epsilon+\epsilon_\delta+\epsilon_c)c & (8\epsilon+2\epsilon_\delta)s \\ -(4\epsilon+\epsilon_\delta)s & (2\epsilon+\epsilon_\delta)c \end{bmatrix} \right\| \cdot \left\| \begin{bmatrix} d_p y_p \\ d_q y_q \end{bmatrix} \right\|,$$

$$(4.31)\quad \left\| \begin{bmatrix} \hat{d_p}'\hat{y_p}' \\ \hat{d_q}'\hat{y_q}' \end{bmatrix} - \begin{bmatrix} d'_p y'_p \\ d'_q y'_q \end{bmatrix} \right\| \leq \left\| \begin{bmatrix} (2\epsilon+\epsilon_\delta+\epsilon_s)s & (8\epsilon+2\epsilon_\delta)c \\ (4\epsilon+\epsilon_\delta)c & -(2\epsilon+\epsilon_\delta)s \end{bmatrix} \right\| \cdot \left\| \begin{bmatrix} d_q y_q \\ d_p y_p \end{bmatrix} \right\|,$$

Remarkably, the r calculation involved no floating point operations, only a direct copy of y_{pi} or y_{qi} for the small and large angle calculations respectively. The above analysis shows that the large angle formulation symmetrically reflects the identical behavior of the small angle formulation. In addition to the bounding of the change of magnitude of the weights at each rotation, the correct selection of rotation respective of angle can minimize ϵ_c, ϵ_s, and ϵ_δ. For extreme angles, e.g. angles generated by rows having one weight much greater than the other, the 2ϵ terms will dominate the error, yielding high accuracy. The above analysis shows that the ordering of weightings is inconsequential. If the scalars are computed with extended precision, the error bound will be tighter at a negligible computational work overhead for larger matrices, as most of the work of the algorithm is in the vector computations.

The following example illustrates the effects of the row sorting in Givens rotations with extreme weights.

Let

$$A(\eta) = \begin{bmatrix} a_{pp} & a_{pq} \\ \eta a_{qp} & \eta a_{qq} \end{bmatrix} \text{ and } \bar{A}(\eta) = \begin{bmatrix} \eta \bar{a}_{pp} & \eta \bar{a}_{pq} \\ \bar{a}_{qp} & \bar{a}_{qq} \end{bmatrix}.$$

where $|a_{ij}| \approx O(1)$, for all i, j. The Givens transformations that insure $a'_{qp} = 0$ and $\bar{a}'_{qp} = 0$ in $A' = G(\eta)A(\eta)$ and $\bar{A}' = \bar{G}(\eta)\bar{A}(\eta)$, respectively, are

$$G(\eta) = \frac{1}{r}\begin{bmatrix} a_{pp} & \eta a_{qp} \\ -\eta a_{qp} & a_{pp} \end{bmatrix} \qquad \bar{G}(\eta) = \frac{1}{\bar{r}}\begin{bmatrix} \eta \bar{a}_{pp} & \bar{a}_{qp} \\ -\bar{a}_{qp} & \eta \bar{a}_{pp} \end{bmatrix}$$

$$r = a'_{pp} = \sqrt{a_{pp}^2 + \eta^2 a_{qp}^2} \quad \text{and} \quad \bar{r} = \bar{a}_{pp} = \sqrt{\eta^2 \bar{a}_{pp}^2 + \bar{a}_{qp}^2}$$

$$\begin{aligned} a'_{pq} &= (a_{pp}a_{pq} + \eta^2 a_{qq}a_{qp})/r & \bar{a}'_{pq} &= (\eta^2 \bar{a}_{pp}\bar{a}_{pq} + \bar{a}_{qq}\bar{a}_{qp})/\bar{r} \\ a'_{qq} &= \eta(a_{pp}a_{qq} - a_{qp}a_{pq})/r. & \bar{a}'_{qq} &= \eta(\bar{a}_{pp}\bar{a}_{qq} - \bar{a}_{qp}\bar{a}_{pq})/\bar{r}. \end{aligned}$$

For sufficiently large $\eta \gg 1$,

$$\begin{aligned} |a'_{pq}| &\approx |\eta a_{qq}| \\ |a'_{qq}| &\approx |a_{pp}a_{qq}/a_{qp} - a_{pq}|, \end{aligned} \quad \text{and} \quad \begin{aligned} |\bar{a}'_{pq}| &\approx |\eta \bar{a}_{pq}| \\ |\bar{a}'_{qq}| &\approx |\bar{a}_{qq} - \bar{a}_{qp}\bar{a}_{pq}/\bar{a}_{pp}|. \end{aligned}$$

For sufficiently small $0 < \eta \ll 1$,

$$\begin{aligned} |a'_{pq}| &\approx |a_{pq}| \\ |a'_{qq}| &\approx |\eta(a_{qq} - a_{qp}a_{pq}/a_{pp})|, \end{aligned} \quad \text{and} \quad \begin{aligned} |\bar{a}'_{pq}| &\approx |\bar{a}_{qq}| \\ |\bar{a}'_{qq}| &\approx |\eta(\bar{a}_{pp}\bar{a}_{qq}/\bar{a}_{qp} - \bar{a}_{pq})|. \end{aligned}$$

In each of the above four cases, information from the row with larger value dominates the resultant superior row.

Also, note that in each case, the more heavily weighted row of the resultant matrix is in the superior position regardless of its initial location. The implication of this property is that a sequence of rotations will move the greater row towards the top of the matrix. This invariance with respect to row ordering implies that Givens QR factorization will perform as though the initial matrix was configured with the greatest row weights superior.

The invariance of the performance of the Givens methods with respect to row ordering may be explained by examining the effect of row order on one rotation. Let one row have an $\mathcal{O}(1)$ scaling and the other have an $\mathcal{O}(\delta), \delta \ll 1$, scaling. Then the formulation for the Givens rotation given in [LH74] yields the following. Let

$$t = \max(|a|, |b|) \qquad u = \min(|a|, |b|) \qquad \omega = u/t \qquad \gamma = \sqrt{1 + \omega^2} \qquad r = t\omega,$$

and let $t = 1$ and $u = \delta$. Then clearly, r is row order invariant, the new top row is $\gamma^{-1}(\mathcal{O}(1) + \delta\mathcal{O}(\delta))$, and the new bottom row is $\gamma^{-1}(-\mathcal{O}(1)\delta + \mathcal{O}(\delta))$ or $\gamma^{-1}(-\mathcal{O}(\delta) + \mathcal{O}(1)\delta)$. The change in the top row is second order in δ and the bottom row is the difference between equally scaled elements for either ordering. The

small angle rotation maintains the effectively greater element superior position in the matrix, whereas the large angle formulation transposes the elements so that the effectively greater element attains the superior position in the matrix. This implies that regardless of the initial ordering of the row norms, the effect is that of having initiated the QR factorization with the rows ordered with the greatest weights superior which is the most stable row ordering.

This analysis is corroborated by our experimental evidence. Additionally, error analysis for the self-scaling fast rotation shows the importance of applying the appropriate large or small angle rotation and rotation parameter calculation, as the use of the incorrect algorithm would violate the bounds and amplify floating point errors.

Table 4.2: **Self-Scaling Fast Rotation Vector Floating Point Intermediate Errors**

$\lvert\theta\rvert \le \frac{\pi}{4},\ \lvert d_p\rvert \ge \lvert d_q\rvert$	$\frac{\pi}{2} > \lvert\theta\rvert > \frac{\pi}{4},\ \lvert d_p\rvert < \lvert d_q\rvert$
$\begin{aligned} \hat{y_p}' &= [y_p + \hat{\beta} y_q(1+\epsilon)](1+\epsilon) \\ &= y_p(1+\epsilon) + \beta y_q(1+5\epsilon) \\ \hat{d_p}'\hat{y_p}' &= \hat{d_p}' y_p(1+\epsilon) \\ &\quad + \hat{d_p}' t \tfrac{d_q}{d_p} y_q(1+5\epsilon) \\ &= c d_p y_p(1+2\epsilon+\epsilon_\delta) \\ &\quad + s d_q y_q(1+6\epsilon+\epsilon_\delta) \end{aligned}$	$\begin{aligned} \hat{y_p}' &= [y_q + \hat{\beta} y_p(1+\epsilon)](1+\epsilon) \\ &= y_q(1+\epsilon) + \beta y_p(1+5\epsilon) \\ \hat{d_p}'\hat{y_p}' &= \hat{d_p}' y_q(1+\epsilon) + \hat{d_p}' t^{-1} \tfrac{d_p}{d_q} y_p(1+5\epsilon) \\ &= s d_q y_q(1+2\epsilon+\epsilon_\delta) \\ &\quad + c d_p y_p(1+6\epsilon+\epsilon_\delta) \end{aligned}$
$\begin{aligned} \hat{y_q}' &= [y_q - \hat{\alpha}\hat{y_p}'(1+\epsilon)](1+\epsilon) \\ &= y_q(1+\epsilon) - \alpha\beta y_q(1+9\epsilon+\epsilon_\delta) \\ &\quad - \alpha y_p(1+5\epsilon+\epsilon_\delta) \\ &= (1-\alpha\beta(1+8\epsilon+\epsilon_\delta)) y_q(1+\epsilon) \\ &\quad - \alpha y_p(1+5\epsilon+\epsilon_\delta) \\ &\quad \{\alpha\beta = \tau\beta/\delta = c^2\tau\beta = c^2 t^2 = s^2\} \\ &= (1-s^2(1+8\epsilon+\epsilon_\delta)) y_q(1+\epsilon) \\ &\quad - \alpha y_p(1+5\epsilon+\epsilon_\delta) \\ &\quad \{\text{let } (1-s^2(1+\epsilon)) = c^2(1+\epsilon_c)\} \\ &= c^2 y_q(1+\epsilon+\epsilon_c) \\ &\quad - cs\tfrac{d_p}{d_q} y_p(1+5\epsilon+\epsilon_\delta) \\ \hat{d_q}'\hat{y_q}' &= c d_q y_q(1+2\epsilon+\epsilon_\delta+\epsilon_c) \\ &\quad - s d_p y_p((1+6\epsilon+2\epsilon_\delta) \end{aligned}$	$\begin{aligned} \hat{y_q}' &= [-y_p + \hat{\alpha}\hat{y_p}'(1+\epsilon)](1+\epsilon) \\ &= -y_p(1+\epsilon) + \alpha\beta y_p(1+9\epsilon+\epsilon_\delta) \\ &\quad + \alpha y_q(1+5\epsilon+\epsilon_\delta) \\ &= -(1-\alpha\beta(1+8\epsilon+\epsilon_\delta)) y_p(1+\epsilon) \\ &\quad + \alpha y_q(1+5\epsilon+\epsilon_\delta) \\ &\quad \{\alpha\beta = \tau\beta/\delta = s^2\tau\beta = s^2 t^{-2} = c^2\} \\ &= -(1-c^2(1+8\epsilon+\epsilon_\delta)) y_p(1+\epsilon) \\ &\quad + \alpha y_q(1+5\epsilon+\epsilon_\delta) \\ &\quad \{\text{let } (1-c^2(1+\epsilon)) = s^2(1+\epsilon_s)\} \\ &= -s^2 y_p(1+\epsilon+\epsilon_c) \\ &\quad + sc\tfrac{d_q}{d_p} y_q(1+5\epsilon+\epsilon_\delta) \\ \hat{d_q}'\hat{y_q}' &= -s d_p y_p(1+2\epsilon+\epsilon_\delta+\epsilon_s) \\ &\quad + c d_q y_q((1+6\epsilon+2\epsilon_\delta) \end{aligned}$

4.2 Numerical Results

4.2.1 Errors in Stiff Matrix QR Factorizations

There are iterative schemes developed for the solution of LSE by WLS using smaller weights [Wam79a, Wam79b, VL85, Bar88]. The convergence of an iterative methods may be accelerated if each iterate is more accurate, thus achieving an error tolerance more rapidly.

In our numerical implementations, we used the fast rotations in two ways–initialized with the diagonal factor matrix D as the square of the weights and also $D = I$ with the weights premultiplied into the matrix. Keeping the squared weights has the advantage that it eliminates the square root operation in the fast rotations. Because the weight matrix is squared, the initial loading of the weights into the D matrix may cause over or underflow problems. A solution to that problem would be to premultiply the problematical weights with their rows. Once set up, it is highly unlikely that the self-scaling fast rotation QR method would under or over flow.

We compare the self-scaling fast rotations with the standard fast plane rotation and with the standard plane rotation, as well as with the Householder's QR method and with the MGS QR method. We used two different elimination orderings combined with merged and non-merged weights for both the standard and self scaling fast rotations, yielding eight fast Givens algorithms. Our computations were performed in Matlab v4 on a Sun Sparc which utilizes IEEE floating-point arithmetic. We tested our algorithms on the three LSE test problems in [VL85] and on the Powell and Reid matrix [PR69]. Following [VL85], we used a large weight $\eta \gg 1$ with B rather than a small weight $\epsilon \ll 1$ with A, and contrasted the accuracy of computing with A *over*

B versus B *over* A to compare the effects of row sorting in different algorithms. The results are expressed graphically in Figures 4.1 – 4.4.

The reference value of x_{LSE} is computed via a generalized singular value decomposition (GSVD) of A and B

$$U^T AX = D_A = \text{diag}(\alpha_1, \ldots, \alpha_n), V^T BX = D_B = \text{diag}(\beta_1, \ldots, \beta_l),$$

where $U \equiv [u_1, \ldots, u_m] \in \mathbb{R}^{m \times m}, V \equiv [v_1, \ldots, v_l] \in \mathbb{R}^{l \times l}$ are orthogonal and $X \equiv [x_1, \ldots, x_n] \in \mathbb{R}^{n \times n}$ is nonsingular, $0 = \alpha_1 = \ldots = \alpha_q < \alpha_{q+1} \leq \ldots \leq \alpha_{l+1} \leq \ldots \leq \alpha_n$, and $\beta_1 \geq \ldots \beta_l > 0$ When

$$y_{\text{LSE}} = \left(\frac{v_1^T d}{\beta_1}, \ldots, \frac{v_l^T d}{\beta_l}, \frac{u_{l+1}^T b}{\alpha_{l+1}}, \ldots, \frac{u_n^T b}{\alpha_n} \right)^T, \tag{4.32}$$

$x_{\text{LSE}} = X y_{\text{LSE}}$ [VL85]. The solution, $x(\eta)$, to the WLS (Equation 3.58), is

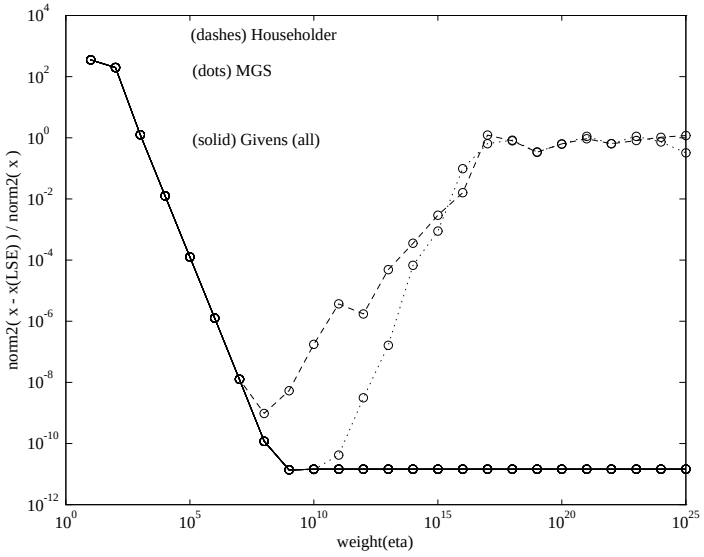
$$x(\eta) = x_{\text{LSE}} + \sum_{i=q+1}^{l} \frac{\gamma_i^2}{\eta^2 + \gamma_i^2} \cdot \frac{\rho_i}{\alpha_i} x_i, \tag{4.33}$$

where $\gamma_i \equiv \frac{\alpha_i}{\beta_i}$, $\rho_i = u_i b - \gamma_i v_i^T d$, and $q = \dim(\mathcal{N}(A)) = \dim(\text{span}(x_1, \ldots, x_q))$.

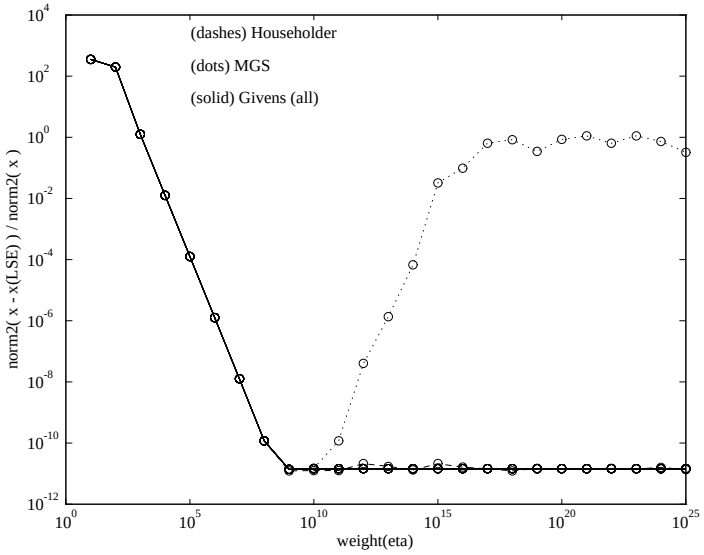
The results we achieved for Householder QR factorization are commensurate with those listed in [VL85] for all three problems. The error, $\|\bar{x}(\eta) - x(\eta)\|_2$, where $\bar{x}(\eta)$ is the computed approximation to $x(\eta)$, displayed in Figure 4.2, shows the Householder's QR method significantly losing accuracy starting at $\eta = 10^7$ for the third test problem ($l = 2, m = 6, n = 4$). The similar (but more absolute) error, $\|\bar{x}(\eta) - x_{\text{LSE}}\|_2$, is displayed in Figure 4.1.

In [BP92], the MGS method is shown to be numerically equivalent to the Householder method applied to a matrix with the $n \times n$ zero matrix adjoined to the top. In our tests, the MGS method also begins significantly losing accuracy starting at

$\eta = 10^{12}$. However, the Givens QR factorizations maintain accuracy for all of the tested η's. Figure 4.3 shows that all of the QR factorization methods give good results for satisfying the constraint equations at all weights. However, both the MGS and Householder methods begin decaying in satisfying the $Ax = b$ equation at $\eta = 10^{16}$ as displayed in Figure 4.4, whereas the Givens methods remain accurate to the most extreme computed weighting. The implication of these data is that to the limit of our tests on these three test matrices, only the Givens QR methods may be employed without the possibility of overshooting the optimum weight when using WLS to solve LSE problems. These tests did not illuminate any significant difference in the accuracies between the different Givens methods and rotation orderings. Similar tests on a sequence of stiff randomly generated matrices with varying structures corroborated these results. The tests examined the results of each of 12 orthogonalization algorithms on 6 random matrices of dimension $\{m = 24, n = 6, l = 4\}$ having 9 different row orderings for each of a wide range of weightings. Additionally, tests on the Powell and Reid matrix in Matlab showed exceptionally good accuracy by the Givens methods for all possible row permutations.

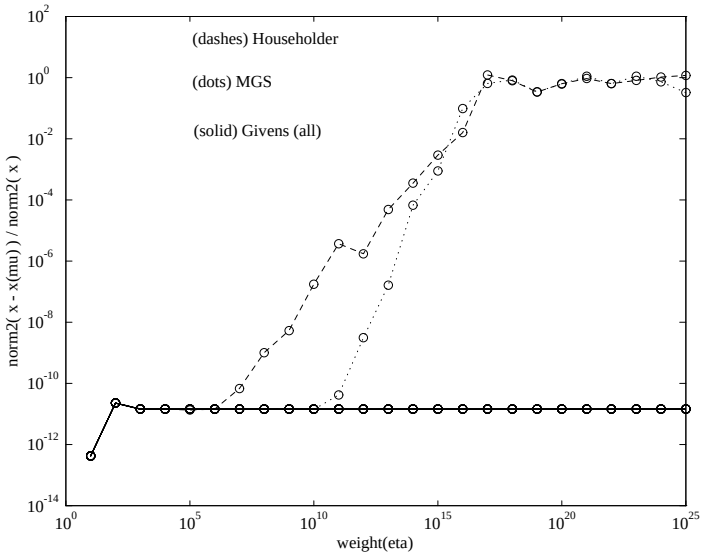


(a) $\min_x \left\| \begin{bmatrix} A \\ \eta B \end{bmatrix} x - \begin{bmatrix} b \\ \eta d \end{bmatrix} \right\|_2, \eta \gg 1$

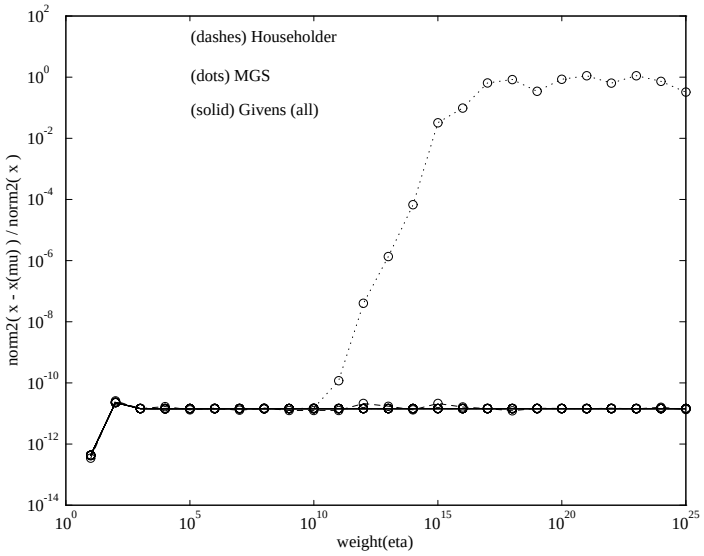


(b) $\min_x \left\| \begin{bmatrix} \eta B \\ A \end{bmatrix} x - \begin{bmatrix} \eta d \\ b \end{bmatrix} \right\|_2, \eta \gg 1$

Figure 4.1: $\|x - x_{LSE}\|_2 / \|x\|_2$: [VL85]'s third matrix w. col. pivoting

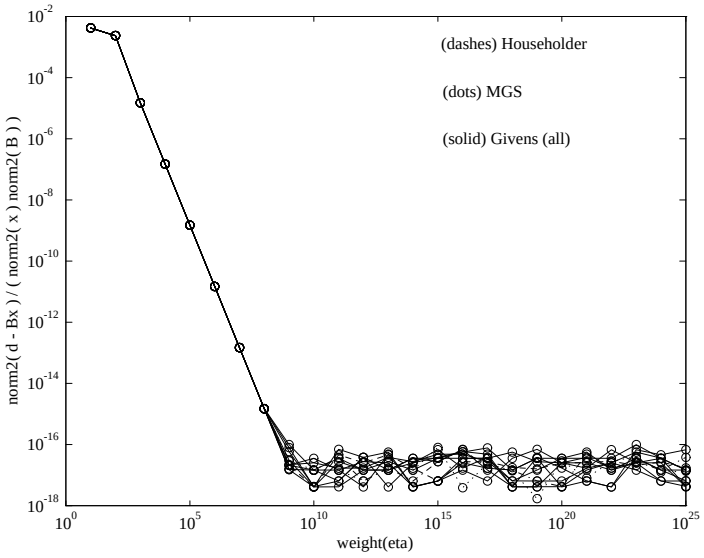


(a) $\min_x \left\| \begin{bmatrix} A \\ \eta B \end{bmatrix} x - \begin{bmatrix} b \\ \eta d \end{bmatrix} \right\|_2, \eta \gg 1$

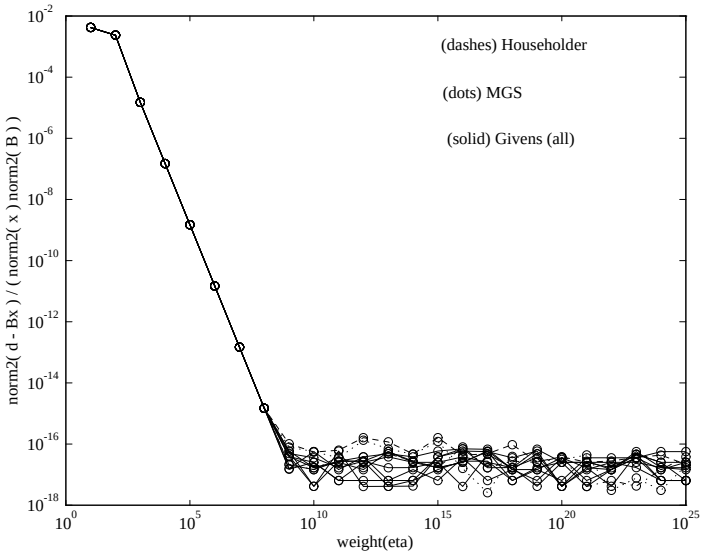


(b) $\min_x \left\| \begin{bmatrix} \eta B \\ A \end{bmatrix} x - \begin{bmatrix} \eta d \\ b \end{bmatrix} \right\|_2, \eta \gg 1$

Figure 4.2: $\|x - x(\eta)\|_2 / \|x\|_2$: [VL85]'s third matrix, w. col. pivoting

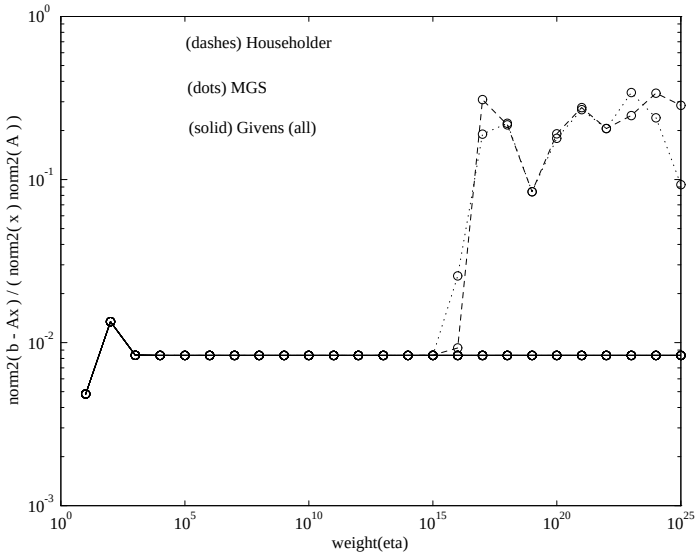


(a) $\min_x \left\| \begin{bmatrix} A \\ \eta B \end{bmatrix} x - \begin{bmatrix} b \\ \eta d \end{bmatrix} \right\|_2, \eta \gg 1$

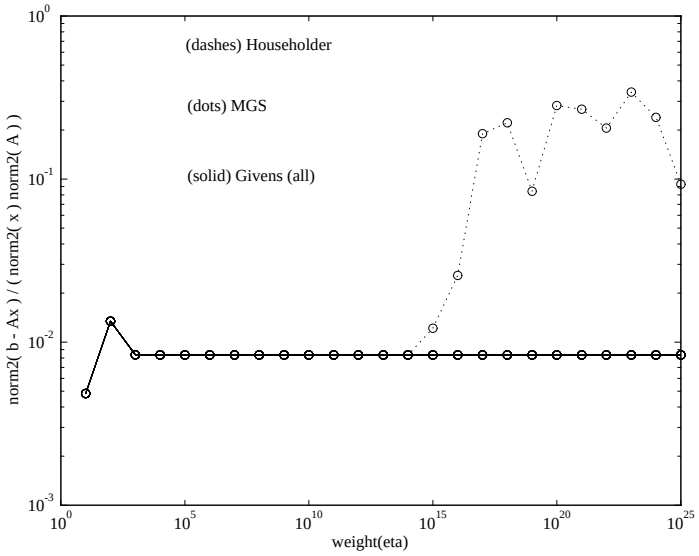


(b) $\min_x \left\| \begin{bmatrix} \eta B \\ A \end{bmatrix} x - \begin{bmatrix} \eta d \\ b \end{bmatrix} \right\|_2, \eta \gg 1$

Figure 4.3: $\|x - x(d - Bx)\|_2 / (\|x\|_2 \|B\|_2)$: [VL85]'s third matrix, w. col. pivoting



(a) $\min_x \left\| \begin{bmatrix} A \\ \eta B \end{bmatrix} x - \begin{bmatrix} b \\ \eta d \end{bmatrix} \right\|_2, \eta \gg 1$



(b) $\min_x \left\| \begin{bmatrix} \eta B \\ A \end{bmatrix} x - \begin{bmatrix} \eta d \\ b \end{bmatrix} \right\|_2, \eta \gg 1$

Figure 4.4: $\|x - x(b - Ax)\|_2 / (\|x\|_2 \|A\|_2)$: [VL85]'s third matrix, w. col. pivoting

Chapter V

EFFICIENCY

5.1 Efficiency on Vector Architectures

In this section, we discuss the architectural features of vector processing units that the vectorized fast plane rotation algorithms can exploit. There are many factors which influence the performance of an algorithm on a vector architecture, e.g., vector length, bank conflicts, and startup time. More general and detailed examinations of these factors can be found in [DDSV91, HJ81, LW89]. The most relevant architectural characteristics for our analysis of fast rotations include:

- The number of functional units and their concurrent operations. We will limit our discussion to only the adder and multiplier units.

- The number and directionality of paths between the register set and the memory.

- Whether the processor is *memory-to-memory* or *register-to-register*. In the latter case, the number of vector registers is relevant.

- Whether functional units can be *chained* together, and whether successive loop iterations can be interleaved.

The *chime*, derived from "*ch*ained vector t*ime*," is proportional to the vector length in a DO loop. Each time an architectural resource such as functional unit, vector register, or memory path, is reused, a new chime commences [DDSV91, LW89], and the number of chimes required for a loop dominates the timing of the loop.

The computational efficiency of the chained fast plane rotation over the standard fast plane rotation results from the elimination of the temporary vector store and read which are necessary for the standard fast plane rotation. In a memory-to-memory architecture, the temporary vector storage and subsequent recall has to be performed explicitly. A hierarchical memory architecture may store and recall the temporary vector from a small fast local or cache memory. A vector register architecture may also store and read from an additional vector in memory, however in most cases, only an additional vector register will be required. Data movement between registers is more efficient than data movement involving memory. Each of the above storage methods ties up the valuable resources of space, time, and communication bandwidth. Memory-to-memory processors, e.g. the ETA, are currently not common. Most vector processors are of the register-to-register class. However, some current supercomputer processor architectures offer hybrid register-to-memory operations which can accept an operand from a register and the other directly from memory to the CPU for dyadic operations[LW89].

We will consider the following simplified representations of three different methods for applying a plane rotation, $\begin{bmatrix} c & -s \\ s & c \end{bmatrix}$, to update the two vectors, w and $v \in \mathcal{R}^n$.

Slow:

$$[\tilde{v}, \tilde{w}] = [v, w] \begin{bmatrix} c & -s \\ s & c \end{bmatrix} = [cv + sw, -sv + cw] \tag{5.1}$$

Standard fast:

$$[\tilde{v}, \tilde{w}] = [v, w] \begin{bmatrix} 1 & \alpha \\ \beta & 1 \end{bmatrix} = [v + \beta w, w + \alpha v] \tag{5.2}$$

Chained fast:

$$[\tilde{v}, \tilde{w}] = [v, w] \begin{bmatrix} 1 & 0 \\ \beta & 1 \end{bmatrix} \begin{bmatrix} 1 & \alpha \\ 0 & 1 \end{bmatrix} = [v + \beta w, w + \alpha(v + \beta w)] \tag{5.3}$$

The best possible timing for a one-sided plane rotation on a vector supercomputer is one chime. Table 5.1 exhibits the minimum number of resources necessary for each rotation to achieve one chime performance. In Jacobi-type algorithms, the ordering of a sequence of plane rotations must be considered for the minimum resource configuration. If two consecutive rotations share a common vector, the memory communication requirement will be halved since it is unnecessary to repeat the read and store of the common vector. Sequential orderings such as the cyclic-by-rows and the cyclic-by-columns orderings maintain a common vector in most of two consecutive stages. Full vector chaining among all vector functional units and memory is required for the one chime performance of any sequence of vector operations and will be assumed in Table 5.1. The register savings of the chained fast rotation could be significant. E.g., if an architecture had 8 registers, and a sufficient number of I/O channels and functional units, twice as many chained rotations could be performed in the register space, as 4 chained fast rotations require only 8 registers, but 3 unchained fast rotations require 9.

5.1.1 Efficiency on Scalar Architectures

At the time this is being written, RISC scalar architectures are overtaking CISC scalar architectures in high performance workstation implementation. Relative to the CISC architectures, RISC architecture performance is more dependent on data flow patterns through the memory hierarchy, from registers, through caches, and memory, and instruction sequencing and scheduling through the instruction pipelines. Whereas, vector based supercomputer architectures could uniformly be relied upon to perform well given a set of vectorizing instructions, the set of RISC architectures is currently more heterogeneous with respect to performance relative to high level coding methodology and guidelines intended to benefit performance. Sources of this variability are variations in page sizes and page swapping heuristics, translation lookaside buffers, cache sizes and architecture, register count and bandwidth between each level of the memory hierarchy, instruction and functional unit architecture, and most significantly, the compiler. If the optimizer determines a way to schedule operations so that the functional units are seldom waiting for operands, the performance will then be high. However a high level coding strategy which is optimal for a particular version of an optimizer for any given scalar architecture may not be optimal for the subsequent release of that optimizer for that architecture. The optimizer must maximize data locality and minimize resource contention via judicious scheduling. Codes having long vectors, up to a rather high point, allow good performance on vector architectures, but not necessarily on scalar architectures, unless the RISC optimizer can find a way to keep the data references from streaming out of virtual dataspace bounds. On a scalar RISC architecture, the memory references must be aggregated and the instructions judiciously scheduled for good performance. We will see in the

benchmarks that follow, the best performing code variants on the vector architecture was definitely not on the scalar RISC architecture and vise versa. A good current strategy for good performance on a scalar RISC processor may to use as many of the highest level vendor or third party library performance tuned subroutines when writing high level code for this architecture.

Table 5.1: **Minimum Single Chime Configuration**

		RESOURCE				
Rotation	Vector	Vector	Memory Paths		Functional Units	
Type	Reuse	Registers	input	output	adder	multiplier
slow	no	3	2	2	2	4
	yes	3	1	1	2	4
slow	no	2	2	2	2	4
chained	yes	2	1	1	2	4
fast	no	3	2	2	2	2
	yes	3	1	1	2	2
fast	no	2	2	2	2	2
chained	yes	2	1	1	2	2

5.2 Benchmark Analysis and Results

The CRAY-X/MP and CRAY-Y/MP series have two memory fetch paths, one memory store path, one adder and one multiplier, and chaining. This is sufficient to

perform a fast chained plane rotation represented in (5.3) in a minimum of two chimes. The CRAY [XY]/MP series and the derivative C90 architecture can perform a vector triad ($z = \alpha x + y$) in one chime since it has (exactly) the minimum number of resources required for a vector triad to perform in one chime. However, for the fast chained rotation, this architecture is deficient by one output memory path and a duplicate set of functional units. Many of the *super-scalar* processors in an emerging set of high-performance workstations are architecturally similar, i.e., optimized for vector triad operations. The CRAY-2, though having a significantly faster clock cycle time, has only one memory path, which is bidirectional but not simultaneously bidirectional, and most critically, has no chaining. The CRAY-2 can perform a fast plane rotation in no less than 6 chimes, though by interleaving the rotations, this figure may approach 5.

5.2.0.1 Cray 2 and [XY]/MP Series

Table 5.2 shows performance figures for both of the CRAY-2 and the CRAY-XM/P for the three rotations represented in (5.1) – (5.3) and for two types of orderings: the cyclic-by-rows ordering which maintains a common vector between two consecutive stages and the round-robin ordering which requires two new vectors for each stage, except for a few exceptional stages. For each matrix order, the CPU times are normalized based on the timing for the round-robin ordering with the standard slow rotation.

5.2.0.2 Cray C90 Series

Additionally, an augmented set of loops and rotations were timed on a Cray C90. Many of those loops were then utilized in a suite of QR factorization algorithms. All programs run on the Crays were expressed in Fortran. The QR factorization benchmark subroutine "qr2b" calculates all the rotation coefficients for a column before applying the respective row rotations. All of the other QR factorization benchmark subroutines apply a row rotation following each calculation of a set of rotation coefficients for a pair of column elements. For a few of the smaller matrix dimensions, these benchmarks were also run on a Sun SPARC workstation having a 40MHz. TI FPU with the Sun f77 Fortran compiler set to optimize with the directives, "-fast -cg89 -O". Additional benchmarks were run on a Sun SuperSPARC FPU with a SuperCache running at 50MHz. with the Sun F77 Fortran compiler set with the directives, "-xcg92 -O4". The results are located in Tables [5.3, 5.4, 5.5, 5.6, 5.7, and 5.9]. The following inference can be made from the Cray loops and QR timings on the C90:

- Inlining subroutine calls in the QR programs resulted in a significant time savings.

- Unrolling (results not shown) of loops surprisingly yielded time penalties.

- Column striping yielded small time penalties unless the stripe was less than the vector length in which case the time penalties were significant.

- The round-robin indexing was performed on the fly and probably was the primary contributory factor in the small time efficiency of the cyclic-by-rows over it.

- Surprisingly, performance was equivalent regardless of whether the data access pattern was in row or column major form.

- The large angle rotations were significantly more efficient than the small angle rotations, due, we infer, to the fact that they can avoid the implicit creation of a temporary vector because they use register renaming to explicitly obviate the temporary. The small angle rotation, however, must create a temporary in the chained case because it has to wait for the add and multiply units to clear. This also helps explain why the chained and unchained loop performances were comparable.

- The computation and application of the rotation parameters in the unchained rotation is simpler and is probably the primary contributor to the small performance advantage of the unchained rotation in the QR timings.

- In the QR timings, the precomputation of the rotation coefficients by column was the most efficient without inlining. However, with inlining, the unchained rotation was fastest, though the adjoined routine came close.

- In almost all cases, the fast rotations were more efficient than the slow rotations.

- Unlike the fast rotation case, the chained slow rotation was significantly more efficient than the non-chained slow rotations.

Remarkably, the SPARC timings yielded a couple of pronounced contrasts to the Cray timings. Whereas the large angle computations were the most efficient of the fast set of rotation algorithms on the Cray, the large angle computations were

the least efficient on the SPARC with the timings being commensurate to the slow rotation algorithms. And, by roughly a factor of three, on the SPARC, the QR factorization algorithm which precomputed entire columns of rotation coefficients was superior, counter to the Cray timing benchmarks. The SuperSPARC with a more recent Fortran compiler yielded significantly different benchmark results. In this case, the large angle penalty was slight, but the penalty for row-wise rather than column-wise operations was large. And, the precomputation of a column's rotation coefficients, subroutine "qr2b", now effected no significant difference from the other orderings. Clearly, one's choice of algorithm should be determined via benchmarking when targeting a contemporary scalar architecture.

Multiples of this minimum configuration will allow loop unrolling or, if a parallel ordering is used, multiple simultaneous rotations. The advantage of the chained fast rotation over the old fast rotation is best realized when one considers a set of eight vector registers. Given that all of the other resource requirements have been met, eight registers can support four of the chained fast rotations but only two of the old fast rotations.

Tailgating, the initiation of a new loop iteration before the previous has completed, also allows efficiency gains. For many architectures, it may be more efficient to precompute the rotation coefficients for an entire block of non-intersecting rotations– thus allowing that part of the computation which had been computed with scalars to be vectorized as well.

Many architectures may allow significant speedup if one of the two vectors remains and is recycled into the subsequent rotation. While this all but rules out the bulk precomputation of the rotation coefficients, it does bump the memory to operation

ratio down to 1/2. This is possible if the QR factorization is being implemented, the i'th row is used as a pivot for the zeroing of the the i'th column, rather than the stepping of contiguous rows. The recycling of vectors is also possible for parallel cyclic orderings for one-sided Jacobi methods– for example, van de Geijn's Leap-Frog ordering in [van89]. In the QR factorization algorithm, if a greedy Givens ordering were used on matrices which were large relative to the vector register length, the rotation parameter computation could be vectorized, and short sequences of contiguous rows could be rotated, allowing for a memory to operation ratio descending to nearly 1/2 versus 1 otherwise.

Table 5.2: **Loop Timing**

(The time scaling differs between matrix sizes)

		Relative CPU Time			
Matrix	Rot.	CRAY 2		CRAY X/MP	
Size	Type	rr	cbr	rr	cbr
64	slow	1.0000	0.8931	0.8465	0.7381
×	standard	0.8668	0.7771	0.7174	0.6188
64	chained	0.8306	0.7191	0.7155	0.6196
128	slow	1.0000	0.9352	0.8855	0.8265
×	standard	0.8703	0.8119	0.7020	0.6724
128	chained	0.8034	0.7352	0.7112	0.6600
256	slow	1.0000	0.9690	0.9115	0.8750
×	standard	0.8458	0.8065	0.7431	0.7287
256	chained	0.8091	0.7656	0.7302	0.6997
512	slow	1.0000	0.9760	0.9126	0.8786
×	standard	0.8425	0.8054	0.7379	0.7186
512	chained	0.7895	0.7717	0.7146	0.7054

Legend:	
slow	The standard slow rotation.
standard	The standard fast rotation.
chained	The chained fast rotation.
cbr	Cyclic-by-rows ordering.
rr	Round-robin ordering.

saxpy1a saxpy with cyclic-by-rows ordering, with the accumulating column changing faster.

saxpy1as Saxpy with cyclic-by-rows ordering, with the accumulating column changing faster, with column segmentation based on vector length.

saxpy1b saxpy with cyclic-by-rows ordering, with the added column changing faster.

saxpy1bs Saxpy with cyclic-by-rows ordering, with the added column changing faster, with column segmentation based on vector length.

saxpy2 saxpy with permuted round-robin ordering

slow1 slow rotation, cyclic-by-rows

slow1s slow rotations with cyclic-by-rows ordering with column segmentation based on vector length.

slow1rs slow rotations with cyclic-by-rows ordering, with rows instead of columns updated, with column segmentation based on vector length.

slowc1 chained slow rotation.

slow2 slow rotation, permuted round-robin

fast1 fast rotation, cyclic-by-rows

fast1s fast rotations with cyclic-by-rows ordering with column segmentation based on vector length.

fast2 fast rotation, permuted round-robin

nfast1 new fast rotation, cyclic-by-rows

nfast1s fast rotations with cyclic-by-rows ordering with column segmentation based on vector length.

nfast1r new fast rotations with cyclic-by-rows ordering, with rows instead of columns updated.

nfast1ls fast rotations with cyclic-by-rows ordering, for large angles, with column segmentation based on vector length.

nfast1rs fast rotations with cyclic-by-rows ordering, with row rather than column updates, with column segmentation based on vector length.

nfast2 new fast rotation, permuted round-robin

nfast2l new fast rotations with permuted round-robin ordering for large angles.

Figure 5.1: LEGEND: Plane Circular Rotation Loops

Table 5.3: **CRAY C916/8512 Loop Timing: (129X128)**

stripe length =	64	128	256
saxpy1a	5.1900E-3	5.2260E-3	5.1454E-3
saxpy1as	6.6315E-3	5.2894E-3	5.2159E-3
saxpy1b	5.2063E-3	5.2250E-3	5.1596E-3
saxpy1bs	6.5244E-3	5.3012E-3	5.1987E-3
saxpy2	7.4167E-3	7.4964E-3	7.4227E-3
slow1	1.2755E-2	1.2793E-2	1.2720E-2
slow1s	1.6355E-2	1.2727E-2	1.2665E-2
slow1rs	1.6439E-2	1.2860E-2	1.2759E-2
slow2	1.4882E-2	1.4939E-2	1.4888E-2
slow1c	1.1409E-2	1.1409E-2	1.1409E-2
fast1	9.2526E-3	9.2930E-3	9.1386E-3
fast1s	1.2121E-2	9.2230E-3	9.1763E-3
fast2	1.1599E-2	1.1585E-2	1.1588E-2
nfast1	9.3123E-3	9.3490E-3	9.2837E-3
nfast1s	1.1827E-2	9.3607E-3	9.2484E-3
nfast1r	9.4521E-3	9.4905E-3	9.4492E-3
nfast1ls	1.0381E-2	8.2048E-3	8.1273E-3
nfast1rs	1.1848E-2	9.2960E-3	9.3033E-3
nfast2	1.1382E-2	1.1321E-2	1.1393E-2
nfast2l	1.0323E-2	1.0288E-2	1.0344E-2

Table 5.4: **CRAY C916/8512 Loop Timing: (257X256)**

stripe length =	64	128	256
saxpy1a	4.0968E-2	4.1041E-2	4.0942E-2
saxpy1as	5.3397E-2	4.2406E-2	4.1291E-2
saxpy1b	4.1266E-2	4.0910E-2	4.0698E-2
saxpy1bs	5.2531E-2	4.1693E-2	4.0844E-2
saxpy2	4.9903E-2	4.9267E-2	4.9554E-2
slow1	1.020E-1	1.012E-1	1.011E-1
slow1s	1.315E-1	1.015E-1	1.005E-1
slow1rs	1.316E-1	1.017E-1	1.007E-1
slow2	1.101E-1	1.096E-1	1.095E-1
slow1c	9.0787E-2	9.0787E-2	9.0786E-2
fast1	7.3129E-2	7.3019E-2	7.2711E-2
fast1s	9.6836E-2	7.3705E-2	7.2284E-2
fast2	8.2400E-2	8.2156E-2	8.2694E-2
nfast1	7.3586E-2	7.3429E-2	7.2573E-2
nfast1s	9.4464E-2	7.3845E-2	7.2880E-2
nfast1r	7.3773E-2	7.3544E-2	7.3492E-2
nfast1ls	8.2915E-2	6.4936E-2	6.4205E-2
nfast1rs	9.4657E-2	7.2463E-2	7.3517E-2
nfast2	8.2553E-2	8.1423E-2	8.1451E-2
nfast2l	7.2949E-2	7.2807E-2	7.2719E-2

Table 5.5: **CRAY C916/8512 Loop Timing: (513X512)**

stripe length =	64	128	256
saxpy1a	0.3267	0.3264	0.3322
saxpy1as	0.4311	0.3403	0.3375
saxpy1b	0.3292	0.3282	0.3311
saxpy1bs	0.4262	0.3353	0.3305
saxpy2	0.3640	0.3607	0.3623
slow1	0.8162	0.8162	0.8171
slow1s	1.0518	0.8207	0.8144
slow1rs	1.0530	0.8209	0.8173
slow2	0.8432	0.8445	0.8472
slow1c	0.7244	0.7244	0.7244
fast1	0.5834	0.5876	0.5984
fast1s	0.7779	0.5902	0.5981
fast2	0.6224	0.6193	0.6292
nfast1	0.5865	0.5924	0.5898
nfast1s	0.7625	0.6005	0.5984
nfast1r	0.5962	0.5870	0.5932
nfast1ls	0.6710	0.5239	0.5200
nfast1rs	0.7585	0.5944	0.5927
nfast2	0.6237	0.6219	0.6199
nfast2l	0.5429	0.5467	0.5515

Table 5.6: **CRAY C916/8512 Loop Timing: (1025X1024)**

stripe length =	64	128	256
saxpy1a	2.6171	2.6370	2.6458
saxpy1as	3.4424	2.7403	2.6849
saxpy1b	2.6125	2.6571	2.6163
saxpy1bs	3.3801	2.7603	2.6772
saxpy2	2.7678	2.8009	2.7576
slow1	6.4686	6.5214	6.5009
slow1s	8.4354	6.5725	6.5347
slow1rs	8.4510	6.5947	6.5363
slow2	6.6145	6.6324	6.6696
slow1c	5.7922	5.8880	6.0476
fast1	4.7183	4.7105	4.6874
fast1s	6.2779	4.7899	4.7559
fast2	4.8522	4.8840	4.8573
nfast1	4.7294	4.7181	4.6958
nfast1s	6.1340	4.8315	4.7391
nfast1r	4.7233	4.7359	4.7226
nfast1ls	5.2506	4.1927	4.1416
nfast1rs	6.1083	4.7699	4.7676
nfast2	4.8977	4.8633	4.8651
nfast2l	4.2684	4.2523	4.2363

Table 5.7: **SPARC + TI FPU @40Mz (-cg89 -fast -O) Loop Timing**

matrix ≡	(65X64)	(129X128)	(257X256)
saxpy1a	17.46	163.12	75.37
saxpy1as	3.80	32.56	75.89
saxpy1b	3.89	30.35	76.00
saxpy1bs	3.84	30.79	75.99
saxpy2	3.93	31.14	78.31
slow1	10.72	87.83	212.56
slow1s	10.66	88.38	214.02
slow1rs	10.78	87.59	219.21
slow2	11.07	88.63	217.47
fast1	7.26	58.77	147.28
fast1s	7.16	58.91	144.05
fast2	7.14	59.72	146.90
nfast1	7.16	58.64	142.23
nfast1s	7.13	57.79	142.60
nfast1r	7.10	58.33	145.80
nfast1ls	10.44	84.47	204.87
nfast1rs	7.12	59.14	146.26
nfast2	7.04	58.20	145.44
nfast2l	10.23	84.71	207.81

Table 5.8: **Super{SPARC+Cache} FPU @50Mz (-xcg92 -O4) Loop Timing**

matrix ≡	(65X64)	(129X128)	(257X256)	(513X512)	(1025X1024)
saxpy1a	0.0277	0.276	1.438	12.28	139.7
saxpy1as	0.0275	0.223	1.408	12.22	133.5
saxpy1b	0.0318	0.190	1.387	11.82	114.0
saxpy1bs	0.0317	0.172	1.392	11.44	103.4
saxpy2	0.0285	0.191	1.517	15.00	172.0
slow1	0.0543	0.333	2.690	22.26	211.3
slow1s	0.0486	0.327	2.667	22.22	210.6
slow1rs	0.0452	0.445	3.296	47.20	823.2
slow2	0.0511	0.347	2.835	25.69	280.1
fast1	0.0386	0.267	1.860	14.61	175.2
fast1s	0.0385	0.270	1.832	15.24	175.6
fast2	0.0422	0.286	2.040	20.31	259.9
nfast1	0.0431	0.290	2.036	17.18	183.5
nfast1s	0.0343	0.297	2.095	17.50	184.4
nfast1r	0.0470	0.629	4.195	57.05	1244.
nfast1ls	0.0383	0.335	2.517	20.60	202.4
nfast1rs	0.0470	0.599	3.052	46.74	733.2
nfast2	0.0366	0.315	2.175	22.58	261.0
nfast2l	0.0411	0.351	2.646	23.68	274.9

Table 5.9: **CRAY C916/8512 QR Factorization Timings** (in place)

Seconds	(129X128)	(257X256)	(513X512)	(1025X1024)
qr0	0.055600	0.2993	1.8223	12.2024
qr1	0.053503	0.2761	1.5926	10.3306
qr2	0.062677	0.3095	1.7451	10.9591
qr2a	0.048891	0.2596	1.5482	10.0881
qr2b	0.033085	0.1926	1.2847	9.1582
With Automatic Subroutine Inlining				
qr0	0.032660	0.2089	1.4606	10.6919
qr1	0.026209	0.1663	1.1667	8.7055
qr2	0.028466	0.1733	1.1910	8.7470
qr2a	0.027301	0.1716	1.1857	8.7057
qr2b	0.032556	0.1926	1.2747	9.0785

Legend

qr0	Slow Givens
qr1	Fast Givens (non-scaling)
qr2	Fast Givens (scaling)
qr2a	Fast Givens (scaling w. Q^T adjoined to L)
qr2b	Fast Givens (scaling w. Rot. coef.'s by cols.)

Table 5.10: **Sun SPARC QR Factorization Timings** (in place)

Seconds	(129X128)	(257X256)	(513X512)	(1025X1024)
SuperSPARC FPU @50Mz +SuperCache (-xcg92 -O4)				
qr0	10.15	81.96	676.8	5502
qr1	9.325	80.84	661.4	5474
qr2	7.860	69.29	569.7	5022
qr2a	8.708	69.68	604.5	4925
qr2b	8.151	71.39	581.7	4922
SPARC + TI FPU @40Mz (-cg89 -fast -O)				
qr0	8.46	68.17		
qr1	7.88	63.29		
qr2	6.84	55.37		
qr2a	9.61	77.24		
qr2b	2.11	16.45		

Legend

qr0	Slow Givens
qr1	Fast Givens (non-scaling)
qr2	Fast Givens (scaling)
qr2a	Fast Givens (scaling w. Q^T adjoined to L)
qr2b	Fast Givens (scaling w. Rot. coef.'s by cols.)

Chapter VI

SUMMARY

We have presented new fast chained plane rotation algorithms which obviate periodic rescaling by dynamically scaling the diagonal factor with one extra comparison per rotation. A further advantage of the chained fast plane rotation results from the elimination of the temporary vector store and read which are necessary in the standard fast plane rotation, though many vector processor architectures lack one or more resources necessary for the full exploitation of the chained fast rotation. As the speed for a floating point vector multiplication approaches that of an addition, as has been the trend for vector processor architectures, the advantage of a fast rotation over the slow rotation, i.e., the elimination of half of the multiplications, becomes less significant; the speed up of what used to be close to 2 is now approximately 3/2. Another advantage of the fast rotation—the elimination of some of the square root computations, is becoming relatively unimportant due to the ability of advanced processors to compute the square root fast. We showed that fast rotations were almost always more efficient than slow rotations at no appreciable loss of accuracy. We additionally presented the minimum architectural attributes necessary for each class of rotation to achieve single chime performance–highlighting the potential computational efficiency advantages of chained plane rotations.

We verified the one-sided Jacobi algorithms of Demmel and Vesselic using fast circular plane rotations. A significant advantage of the one-sided Jacobi algorithm

over the two-sided algorithm is that the memory references are mostly contiguous. The benefit comes from the fact that unit stride vector reference is faster than gather and scatter. Also memory bank conflicts are minimized by unit stride reference. Although on the C90, this is much less of an issue if the strides are of an odd order. However, we would still recommend performing complex vector computational linear algebra by storing complex vectors as real vectors adjoined by imaginary vectors rather than as contiguous singleton pairs. Equivalently, in the context of the QR factorization, vector lengths may be increased by adjoining the matrix, its right-hand-side, and its orthogonal matrix.

We described the application of plane rotations towards the solution and single row update and downdate of least squares problems. The equivalence of LINPACK downdating, hyperbolic rotation downdating, and fast negatively weighted rotation downdating was shown.

It is essential, for scaling and the numerical stability of fast rotations, to choose the appropriate rotation based on the relative size of the angle with $\pm\pi/4$. However, certain algorithms do not require rotation angles to exceed $[-\pi/4, \pi/4]$, simplifying the choices in fast rotations. We have shown that the scaling of the chained fast rotation is highly dependent upon the direction of the chaining, i.e., which of the two vectors uses the updated value of the other vector for its own update. Although the 4-way branch algorithm (Table 2.1) yields the better scaling, as no diagonal element will be pushed away from unity in any two consecutive rotations, the 2-way branch algorithm (Table 2.2) is simpler and more efficient, and it provides excellent control of scaling. We have not found an instance for which the 2-way branch algorithm performs poorly in scaling of the diagonal factor.

Contradicting some statements in the literature, we showed that both fast and standard Givens rotation-based algorithms produce accurate results regardless of row sorting with even extremely large weights when equality constrained least squares problems are solved by the weighting method. However, we also showed the necessity of column sorting. We also produced evidence which contradicted some statements in the literature which asserted that the modified Gram-Schmidt algorithm is insensitive to row sorting. We found it to be a little less sensitive than Householder factorization. We presented the results of our numerical experiments which showed that for a large spread of row weights, self-scaling fast Givens rotations exhibit superior accuracy to Householder and modified Gram-Schmidt QR factorizations for arbitrary row orderings. The numerical results showed that once a high accuracy was achieved, the self-scaling rotations maintained that accuracy for much larger weights, thus obviating the need for iterating with insufficient row weights as described in [VL85].

BIBLIOGRAPHY

[ADdR89] M. Arioli, I. S. Duff, and P. P. M. de Rijk. On the augmented system approach to sparse least-squares problems. *Numerische Mathematik*, 55:667–684, 1989.

[AP89] A. A. Anda and H. Park. Jacobi algorithms for non-symmetric eigenvalue problems on a CRAY-2. presented at the *Fourth SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, Il., December 1989.

[AP92] A. A. Anda and H. Park. Fast computation of eigenvalue decompositions on vector architectures. In P. M. Pardalos, editor, *Advances in Optimization and Parallel Computing*, pages 26–41. North-Holland, 1992.

[AP93] A. A. Anda and H. Park. Fast QR decomposition for weighted least squares problems. In *Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing*, pages 399–404, 1993.

[AP94a] A. A. Anda and H. Park. Fast plane rotation algorithms with dynamic scaling. *SIAM J. Matrix Anal. Appl.*, 15(1):162–174, Jan 1994.

[AP94b] A. A. Anda and H. Park. Self-scaling fast rotations for stiff least squares problems. Accepted to LAA, 1994, 1994.

[APP88] S. T. Alexander, C. T. Pan, and R. J. Plemmons. Analysis of a recursive least squares hyperbolic rotation algorithm for signal processing. *Lin. Alg. and Its Applic.*, 98:3–40, 1988.

[Bar85] J. L. Barlow. Stability analysis of the G-algorithm and a note on its application to sparse least squares problems. *BIT*, 25:507–520, 1985.

[Bar88] J. Barlow. Error analysis and implementation aspects of deferred correction for equality constrained least squares problems. *SIAM J. Numer. Anal.*, 25(6):1340–1358, Dec 1988.

[BBvd87] A. W. Bojanczyk, R. P. Brent, P. van Dooren, and F. R. de Hoog. A note on downdating the Cholesky factorization. *SIAM J. Sci. Stat. Comput.*, 8(3):210–221, May 1987.

[BCH91] C. Bischof and P. Christian Hansen. Structure-preserving and rank-revealing QR- factorizations. *SIAM J. Sci. Stat. Comput.*, 12(6):1332–1350, November 1991.

[BD80] Å. Björck and I. S. Duff. A direct method for the solution of sparse linear least squares problems. *laa*, 34:43–68, 1980.

[BG81] A. Bunse-Gerstner. An analysis of the HR algorithm for computing the eigenvalues of a matrix. *Lin. Alg. and Its Applic.*, 35:155–173, 1981.

[BH88] J. Barlow and S. Handy. The direct solution of weighted and equality constrained least-squares problems. *SIAM J. Sci. Stat. Comput.*, 9(4):704–716, Jul 1988.

[BI87] J. Barlow and I. Ipsen. Scaled Givens rotations for the solution of linear least squares problems on systolic arrays. *SIAM J. Sci. Stat. Comput.*, 8(5):716–733, sep 1987.

[Bjö67] Å. Björck. Solving linear least squares problems by Gram–Schmidt orthogonalization. *BIT*, 7:1–21, 1967.

[Bjö84] Å. Björck. A general updating algorithm for constrained linear least squares problems. *SIAM J. Sci. Stat. Comput.*, 5(2):394–402, jun 1984.

[Bjö90] Å. Björck. Least squares methods. In P. G. Ciarlet and J. L. Lions, editors, *Handbook of Numerical Analysis, V.1: Solution of Equations in* R^n, pages 465–652. Elsevier Science Publishers B. V. (North-Holland), 1990.

[Bjö91] Å. Björck. Algorithms for linear least squares problems. Technical report, Department of Mathematics, Linköping University, August 1991.

[BL85] R. P. Brent and F. T. Luk. The solution of singular-value and symmetric eigenvalue problems on multiprocessor arrays. *SIAM J. Sci. Stat. Comput.*, 6(1):69–84, January 1985.

[BLVL85] R. P. Brent, F. T. Luk, and C. Van Loan. Computation of the singular value decomposition using mesh-connected processors. *Journal of VLSI and Computer Systems*, 1(3):242–270, 1985.

[BNP88] J. L. Barlow, N. K. Nichols, and R. J. Plemmons. Iterative methods for equality constrained least squares problems. *SIAM J. Sci. Stat. Comput.*, 9(5):892–906, Sep 1988.

[BP92] Å. Björck and C. C. Paige. Loss and recapture of orthogonality in the modified Gram-Schmidt algorithm. *SIAM J. Matrix Anal. Appl.*, 13:176–190, 1992.

[BPE94] Å. Björck, H. Park, and L. Eldén. Accurate downdating of least squares solutions. *SIAM J. Matrix Anal. Appl.*, 15(2):549–568, April 1994.

[BS89] A. W. Bojanczyk and A. O. Steinhardt. Stabilized hyperbolic Householder transformations. *IEEE Trans. Acoust., Speech, Signal Processing*, 37(8):1286–1288, 1989.

[CB90] G. Cybenko and M. Berry. Hyperbolic Householder algorithms for factoring structured matrices. *SIAM J. Matrix Anal. Appl.*, 11(4):499–520, October 1990.

[CKLA87] J. Chun, T. Kailath, and H. Lev-Ari. Fast parallel algorithms for QR and triagular factorization. *SIAM J. Sci. Stat. Comput.*, 8(6):899–913, November 1987.

[Dax83] A. Dax. A diagonal modification for the downdating algorithm. *SIAM J. Sci. Stat. Comput.*, 4(1):85–93, March 1983.

[DBMS78] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users Guide*. SIAM, Philadelphia, PA, 1978.

[DDSv89] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. Linear algebra computations on vector and parallel computers. Short Course text from Fourth SIAM Conference on Parallel Processing for Scientific Computing, Chicago., December 1989.

[DDSV91] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. Van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. SIAM, Philadephia, 1991.

[DE84] J. J. Dongarra and S. C. Eisenstat. Squeezing the most out of an algorithm in CRAY fortran. *ACM Transactions on Mathematical Software*, 10(3):219–230, September 1984.

[DGK84] J. J. Dongarra, F. G. Gustavson, and A. Karp. Implementing linear algebra algorithms for dense matrices on a vector pipeline machine. *SIAM Review*, 26(1):91–112, January 1984.

[DGKS76] J. Daniel, W. B. Gragg, L. Kaufman, and G. W. Stewart. Reorthogonalization and stable algorithms for updating the Gram-Schmidt QR factorization. *Math. Comp.*, 30:772–795, 1976.

[DI86] J. Delosme and I. C. F. Ipsen. Parallel solution of symmetric positive definite systems with hyperbolic rotations. *Lin. Alg. and Its Applic.*, 77:75–111, 1986.

[dR89] P. de Rijk. A one-sided Jacobi algorithm for computing the singular value decomposition on a vector computer. *SIAM J. Sci. Stat. Comput.*, 10(2):359–371, mar 1989.

[DS86] J. J. Dongarra and D. C. Sorensen. Linear algebra on high-performance computers. In M. Feilmeier, G. Joubert, and U. Schendel, editors, *Parallel Computing 85*, pages 3–32, New York, 1986. Second International Conference on Parallel Computing '85, North Holland.

[DV88] J. Demmel and K. Veselić. Jacobi's method is more accurate than QR. Technical report, Courant Institute of Mathematical Sciences, Department of Computer Science, New York University, 1988.

[Ebe87] P. J. Eberlein. On the Schur decomposition of a matrix for parallel computation. *IEEE Transactions on Computers*, C-36(2):167–174, February 1987.

[EGK91] S. Elhay, G. Golub, and J. Kautsky. Updating and downdating of orthogonal polynomials with data fitting applications. *SIAM J. Matrix Anal. Appl.*, 12(2):327–353, April 1991.

[Eld77] L. Eldén. Algorithms for the regularization of ill-conditioned least squares problems. *BIT*, 17:134–145, 1977.

[Eld80] L. Eldén. Perturbation theory for the least squares problem with linear equality constraints. *SIAM J. Numer. Anal.*, 17(3):338–350, 1980.

[Eld82] L. Eldén. A weighted pseudoinverse, generalized singular values, and constrained least squares problems. *BIT*, 22:487–502, 1982.

[EP90] P. J. Eberlein and H. Park. Efficient implementation of Jacobi algorithms and Jacobi sets on distributed memory architectures. *J. of Par. and Dist. Comput., special issue on Algorithms for Hypercube Computers*, 8:358–366, 1990.

[EP94] L. Eldén and H. Park. Perturbation analysis for block downdating of a Cholesky decomposition. *Numer. Math.*, 68:457–467, 1994.

[FH60] G. E. Forsythe and P. Henrici. The cyclic Jacobi method for computing the principal values of a complex matrix. *Trans. Amer. Math. Soc.*, 94:1–23, 1960.

[Gen73a] W. M. Gentleman. Error analysis of QR decompositions by Givens transformations. *Lin. Alg. and Its Applic.*, 10:189–197, 1973.

[Gen73b] W. M. Gentleman. Least squares computations by Givens transformations without square roots. *J. Inst. Maths Applics*, 13:329–336, 1973.

[GGMS84] P. E. Gill, G. H. Golub, W. Murray, and M. A. Saunders. Methods for modifying matrix factorizations. *Math. Comp.*, 28(126):505–535, April 1984.

[Gol69] G. H. Golub. Matrix decompositions and statistical computation. In R. C. Milton and J. A. Nelder, editors, *Statistical Computation*, pages 365–397. Academic Press, New York, 1969.

[GPS90] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1):54–135, March 1990.

[Gre55] J. Greenstadt. A method for finding roots of arbitrary matrices. *Math. Tables Aids Comput.*, 9(50):47–52, 1955.

[GS91] J. Gotze and U. Schwiegelshohn. A square root and division free Givens rotation for solving least squares problems on systolic arrays. *SIAM J. Sci. Stat. Comput.*, 12(4):800–807, July 1991.

[GVL89] G. H. Golub and C. F. Van Loan. *Matrix Computations.* Johns Hopkins Series in the Mathematical Sciences. Johns Hopkins Press, Baltimore, MD, second edition, 1989.

[GW92] M. Gulliksson and P.-Å. Wedin. Modifying the QR-decomposition to constrained and weighted linear least squares. *SIAM J. Matrix Anal. Appl.*, 13(4):1298–1313, Oct 1992.

[Hag88] W. W. Hager. *Applied Numerical Linear Algebra.* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[Ham74] S. Hammarling. A note on modifications to the Givens plane rotation. *J. Inst. Maths Applics*, 13:215–218, 1974.

[Hes58] M. R. Hestenes. Inversion of matrices by biorthogonalization and related results. *J. SIAM*, 6:51–90, 1958.

[HJ81] R. Hockney and C. Jesshope. *Parallel Computers: Architecture, Programming and Algorithms.* Adam Hilger, Ltd., Bristol, UK, 1981.

[Kah66] W. Kahan. Numerical linear algebra. *Can. Math. Bull.*, 9:757–801, 1966.

[LH74] C. L. Lawson and R. J. Hanson. *Solving Least Squares Problems.* Prentice Hall Series in Automatic Computation. Prentice Hall, Englewood Cliffs, NJ, 1974.

[LHKK79a] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Algorithm 539: Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5:324–325, 1979.

[LHKK79b] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5:308–323, 1979.

[LP89] F. T. Luk and H. Park. On parallel Jacobi orderings. *SIAM J. Sci. Stat. Comput.*, 10(1):18–26, January 1989.

[LW70] Ö. Leringe and P.-Å. Wedin. A comparison between different methods to compute a vector x which minimizes $\|Ax - b\|$, when $Gx = h$. Technical report, Department of Computer Science, Lund University, 1970.

[LW89] J. M. Levesque and J. W. Williamson. *A Guidebook to Fortran on Supercomputers*. Academic Press, San Diego, CA, 1989.

[Ort88] J. M. Ortega. *Introduction to Parallel and Vector Solution of Linear Systems*. Frontiers of Computer Science. Plenum Press, New York, 1988.

[Pan90] C.-T. Pan. A modification to the LINPACK downdating algorithm. *BIT*, 30:707–722, 1990.

[Par80] B. N. Parlett. *The Symmetric Eigenvalue Problem*. Prentice Hall Series in Computational Mathematics. Prentice Hall, Englewood Cliffs, N. J., first edition, 1980.

[PE90] H. Park and P. J. Eberlein. Factored Jacobi-like algorithms for eigensystem computations on vector processors. Technical Report TR90-11, Computer Science Dept., Univ. of Minnesota, February 1990.

[PH93] H. Park and V. Hari. A real algorithm for the Hermitian eigenvalue decomposition. *BIT*, 33:158–171, 1993.

[PR69] M. Powell and J. Reid. On applying Householder's method to linear least squares problems. In A. Morell, editor, *Proceedings IFIP Congress 1968*, pages 122–126, Amsterdam, 1969. North-Holland.

[PW70] G. Peters and J. H. Wilkinson. The least squares problem and pseudo-inverses. *The Computer Journal*, 13:309–316, 1970.

[Rat82] W. Rath. Fast Givens rotations for orthogonal similarity transformations. *Numer. Math.*, 40:47–56, 1982.

[RS86] C. M. Rader and A. O. Steinhardt. Hyperbolic Householder transformations. *IEEE Trans. Acoust., Speech, Signal Processing*, 34(6):1589–1602, December 1986.

[SS79] K. Schittkowski and J. Stoer. A factorization method for the solution of constrained linear least squares problems allowing for subsequent data changes. *Numer. Math.*, 31:431–463, 1979.

[Ste76] G. W. Stewart. The economical storage of plane rotations. *Numer. Math.*, 25:137–138, 1976.

[Ste79] G. W. Stewart. The effects of rounding error on an algorithm for downdating a Cholesky factorization. *J. Inst. Maths Applics*, 23:203–213, 1979.

[Ste84] G. W. Stewart. On the asymptotic behavior of scaled singular value and QR decompostions. *Math. Comp.*, 43(168):483–489, Oct 1984.

[Ste85] G. W. Stewart. A Jacobi-like algorithm for computing the Schur decomposition of a nonhermitian matrix. *SIAM J. Sci. Stat. Comput.*, 6(4):853–864, October 1985.

[van89] R. A. van de Geijn. Leap-frog ordering for parallel one-sided Jacobi methods. presented at the *Fourth SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, Il., December 1989.

[Vav92] S. A. Vavasis. Stable numerical algorithms for equilibrium systems. Technical Report 92-1280, Department of Computer Science, Cornell University, Ithaca, NY, 1992. Accepted to *SIAM J. Matrix Analysis*.

[VH89] K. Veselić and V. Hari. A note on the one-sided Jacobi algorithm. *Numer. Math.*, 56:627–633, 1989.

[VL83] C. F. Van Loan. A generalized SVD analysis of some weighting methods for equality constrained least squares. In Pite Havsbad, B. Kågström, and A. Ruhe, editors, *Matrix Pencil Proceedings, Lecture Notes in Mathematics 973*, pages 245–262, New York, 1983. Springer-Verlag.

[VL85] C. F. Van Loan. On the method of weighting for equality constrained least squares problems. *SIAM J. Numer. Anal.*, 22:851–864, 1985.

[Wam79a] R. H. Wampler. L2A and L2B, weighted least squares solutions by modified Gram–Schmidt with iterative refinement. *ACM Trans. Math. Soft.*, 5(4):494–499, dec 1979.

[Wam79b] R. H. Wampler. Solutions to weighted least squares problems by modified Gram–Schmidt with iterative refinement. *ACM Trans. Math. Soft.*, 5(4):457–465, dec 1979.

Appendix I

GIVENS ROTATION BASED QR FACTORIZATIONS IN MATLAB

A.1 QR Factorization Using Standard Givens Rotations

A.1.1 Matlab function gqr1.m

```
function [Q,R] = gqr1(A)
% Form the QR decomposition by Givens transformations.
% [Q,R] = gqr1(A)
%
% input
%    A          the rectangular matrix to undergo orthogonal
%               decompositon
% output
%    Q          the orthogonal factor
%    R          the upper triangular factor
% calls
%    givens
%    rowrot
%
[m,n]=size(A);
Q=eye(m);
for j=1:n-1
    for i=m:-1:j+1
        [c,s,r]=givens1(A(i-1,j),A(i,j));
        A(:,j+1:n)=rowrot(A(:,j+1:n),c,s,i-1,i);
        Q=colrot(Q,c,s,i-1,i);
        A(i-1,j)=r;
        A(i,j)=0;
    end
end
% rotate out the bottom m-n elements in the last column if m>n
for i=m:-1:n+1
   [c,s,r]=givens1(A(i-1,n),A(i,n));
```

```
   Q=colrot(Q,c,s,i-1,i);
   A(i-1,n)=r;
   A(i,n)=0;
end
R=A;
```

A.1.2 Matlab function **givens1.m**

function `[c,s,r] =` **givens1**`(a,b)`

```
%
%             [c,s]=givens(a,b)
%
%        This subroutine finds c and s such that
%           /c -s\/a\ _ /r\
%           \s  c/\b/ - \0/
%
%      The algorithm used is given by 5.1.5 in Matrix Computations
%      by Golub and Van Loan, and Algorithm G1 in Solving Least
%      Squares Problems by Lawson and Hanson.
%

if  b == 0
   c=1.0;
   s=0.0;
   r=a;
elseif a == 0
   c=0.0;
   s=1.0;
   r=b;
else
   if  abs(b) > abs(a)
      t=-a/b;
      u=sqrt(1.0+t*t);
      r=b*u;
      s=-1.0/u;
      c=s*t;
   else
      t=-b/a;
      u=sqrt(1.0+t*t);
      r=a*u;
      c=1.0/u;
```

```
      s=c*t;
   end
end
```

A.1.3 Matlab function **colrot.m**

```
function A = colrot(A,c,s,p,q)
%
% This subroutine performs a Givens rotation which
% overwrites A with A G(c,s).
%
% input
%    A          the matrix undergoing rotation (in place)
%    c,s        the cosine and sine of the rotation angle
%    p,q        the top and bottom row indices for the rotation
% output
%    A          the matrix rotated in place
%
% (Based on the code fragment atop page 216 of the "Handbook for
%  Matrix Computations" by Coleman and Van Loan)
%
A(:,[p q])=A(:,[p q])*[c s;-s c];
```

A.1.4 Matlab function **rowrot.m**

```
function A = rowrot(A,c,s,p,q)
%
% This subroutine performs a Givens rotation which
% overwrites A with G(c,s)^T A.
%
% input
%    A          the matrix undergoing rotation (in place)
%    c,s        the cosine and sine of the rotation angle
%    p,q        the top and bottom row indices for the rotation
% output
%    A          the matrix rotated in place
%
% (Based on the code fragment atop page 216 of the "Handbook for
%  Matrix Computations" by Coleman and Van Loan)
%
A([p q],:)=[c -s;s c]*A([p q],:);
```

A.2 QR Factorization Using Standard Fast Givens Rotations

A.2.1 Matlab function **fgqr1.m**

```
function [Q,R,d] = fgqr1(A,d)
% Form the QR decomposition by fast Givens transformations.
% [Q,R,d] = fgqr1(A,d)
%
% input
%    A          the rectangular matrix to undergo orthogonal
%               decompositon
%    d          the diagonal factor
% output
%    Q          the orthogonal factor
%    R          the upper triangular factor
%    d          the diagonal factor
% calls
%    f1g
%    f1rowrot
%    f1colrot
%
[m,n]=size(A);
if nargin == 1
    d=ones(m,1);
else
    d=d.^2;
end
Q=eye(m);
for j=1:n-1
    for i=m:-1:j+1
        [alpha,beta,r,d([i-1 i]),case]=f1g(A(i-1,j),A(i,j),d([i-1 i]));
        A(:,j+1:n)=f1rowrot(A(:,j+1:n),alpha,beta,i-1,i,case);
        Q=f1colrot(Q,alpha,beta,i-1,i,case);
        A(i-1,j)=r;
        A(i,j)=0;
    end
end
% rotate out the bottom m-n elements in the last column if m>n
for i=m:-1:n+1
   [alpha,beta,r,d([i-1 i]),case]=f1g(A(i-1,n),A(i,n),d([i-1 i]));
```

```
   Q=f1colrot(Q,alpha,beta,i-1,i,case);
   A(i-1,n)=r;
   A(i,n)=0;
end
for i=1:min([m n])
    if d(i) < 0
        d(i) = 0;
    end
    sqrtdi=sqrt(d(i));
    R(i,:)=A(i,:)*sqrtdi;
    Q(:,i)=Q(:,i)*sqrtdi;
end
for i=n+1:m
    if d(i) < 0
        d(i) = 0;
    end
    R(i,:)=A(i,:);
    Q(:,i)=Q(:,i)*sqrt(d(i));
end
```

A.2.2 Matlab function fg1.m

```
function [alpha,beta,r,d,case] = f1g(a,b,d)

%
%             [alpha,beta]=fast_givens(a,b,d)
%
%        This subroutine finds beta and alpha such that:
%
%        for case==2 (small angle)
%           /1   beta\/a\ _ /r\ , /d(1)\ _ /d(1)*cos\
%           \alpha  1/\b/ - \0/   \d(2)/ - \d(2)*cos/
%
%        else for case==3 (large angle)
%           /beta   1\/a\ _ /r\   /d(1)\ _ /d(2)*sin\
%           \-1 alpha/\b/ - \0/ , \d(2)/ - \d(1)*sin/
%
%      The algorithm used is given based on the algorithm on pages
%      60-61 Solving Least Squares Problems by Lawson and Hanson.
%

if b == 0
```

```
   case=1;

   beta=0.0;
   alpha=0.0;
   r=a;
else
       alpha=b/a;
       gamma=d(1)/d(2);
       if alpha^2 <= gamma

       case=2;

       beta=alpha/gamma;
       delta=1+beta*alpha;
       alpha=-alpha;
       r=a*delta;
       d(2)=d(2)/delta;
       d(1)=d(2)*gamma;

    else

       case=3;

       alpha=1/alpha;
       beta=alpha*gamma;
       delta=1+beta*alpha;
       r=b*delta;
       d(1)=d(2)/delta;
       d(2)=d(1)*gamma;
    end
end
```

A.2.3 Matlab function f1colrot.m

```
function A = f1colrot(A,alpha,beta,p,q,case)
% A = f1colrot(A,alpha,beta,p,q,case)
%
% This subroutine performs a fast Givens rotation which
% overwrites A with A F(c,s).
%
```

```
% input
%    A          the matrix undergoing rotation (in place)
%    alpha      the q'th row saxpy factor
%    beta       the p'th row saxpy factor
%    p,q        the top and bottom row indices for the rotation
%    case       flag indicating the angle based rotation type
% output
%    A          the matrix rotated in place
%
if  case == 2
%   (small angle)
    tmp=A(:,p);
    A(:,p)=A(:,p)+beta*A(:,q);
    A(:,q)=A(:,q)+alpha*tmp;
elseif case == 3
%   (large angle)
    tmp=A(:,p);
    A(:,p)=A(:,q)+beta*A(:,p);
    A(:,q)=alpha*A(:,q)-tmp;
end
```

A.2.4 Matlab function **f1rowrot.m**

```
function A =  f1rowrot(A,alpha,beta,p,q,case)
% A = f1rowrot(A,alpha,beta,p,q,case)
%
% This subroutine performs a fast Givens rotation which
% overwrites A with F(c,s)^T A.
%
% input
%    A          the matrix undergoing rotation (in place)
%    alpha      the q'th row saxpy factor
%    beta       the p'th row saxpy factor
%    p,q        the top and bottom row indices for the rotation
%    case       flag indicating the angle based rotation type
% output
%    A          the matrix rotated in place
%
if  case == 2
%   (small angle)
    tmp=A(p,:);
```

```
    A(p,:)=A(p,:)+beta*A(q,:);
    A(q,:)=A(q,:)+alpha*tmp;
elseif case == 3
%   (large angle)
    tmp=A(p,:);
    A(p,:)=A(q,:)+beta*A(p,:);
    A(q,:)=alpha*A(q,:)-tmp;
end
```

A.3 QR Factorization Using Scaled Fast Givens Rotations

A.3.1 Matlab function fgqr2.m

```
function [Q,R,d] = fgqr2(A,d)
% [Q,R,d] = fgqr2(A,d)
% Form the QR decomposition by fast scaled Givens transformations.
%
% input
%    A          the rectangular matrix to undergo orthogonal
%               decompositon
%    d          the diagonal factor
% output
%    Q          the orthogonal factor
%    R          the upper triangular factor
%    d          the diagonal factor
% calls
%    f2g
%    f2rowrot
%    f2colrot
%
[m,n]=size(A);
if nargin == 1
    d=ones(m,1);
else
    d=d.^2;
end
Q=eye(m);
for j=1:n-1
    for i=m:-1:j+1
        [alpha,beta,r,d([i-1 i]),case,flow]=f2g(A(i-1,j),A(i,j),d([i-1 i]));
        A(:,j+1:n)=f2rowrot(A(:,j+1:n),alpha,beta,i-1,i,case,flow);
        Q=f2colrot(Q,alpha,beta,i-1,i,case,flow);
        A(i-1,j)=r;
        A(i,j)=0;
    end
end
% rotate out the bottom m-n elements in the last column if m>n
for i=m:-1:n+1
   [alpha,beta,r,d([i-1 i]),case,flow]=f2g(A(i-1,n),A(i,n),d([i-1 i]));
   Q=f2colrot(Q,alpha,beta,i-1,i,case,flow);
   A(i-1,n)=r;
```

```
   A(i,n)=0;
end
for i=1:min([m n])
    if d(i) < 0
        d(i) = 0;
    end
    sqrtdi=sqrt(d(i));
    R(i,:)=A(i,:)*sqrtdi;
    Q(:,i)=Q(:,i)*sqrtdi;
end
for i=n+1:m
    if d(i) < 0
        d(i) = 0;
    end
    R(i,:)=A(i,:);
    Q(:,i)=Q(:,i)*sqrt(d(i));
end
```

A.3.2 Matlab function fg2.m

function [alpha,beta,r,d,case,flow] = **f2g**(a,b,d)

```
%
%             [alpha,beta,d]=scaled_fast_givens(a,b,d)
%
%        This subroutine finds beta and alpha such that:
%
%        for case==1
%           No rotation
%
%        for case==2, flow==1 (small angle, d(1)>=d(2))
%           /1      0\/1  beta\/a\ _ /r\ , /d(1)\ _ /d(1)*cos\
%           \alpha  1/\0     1/\b/ - \0/   \d(2)/ - \d(2)/cos/
%
%        for case==3, flow==1 (large angle, d(1)>=d(2))
%           /1   beta\/0      1\/a\ _ /r\   /d(1)\ _ /d(2)/sin\
%           \0      1/\-1 alpha/\b/ - \0/ , \d(2)/ - \d(1)*sin/
%
%        for case==2, flow==2 (small angle, d(1)<d(2))
%           /1   beta\/1      0\/a\ _ /r\ , /d(1)\ _ /d(1)/cos\
%           \0      1/\alpha  1/\b/ - \0/   \d(2)/ - \d(2)*cos/
%
```

```
%        for case==3, flow==2 (large angle, d(1)<d(2))
%           /1      0\/beta   1\/a\ _ /r\   /d(1)\ _ /d(2)*sin\
%           \alpha -1/\1      0/\b/ - \0/ , \d(2)/ - \d(1)/sin/
%
%      The algorithm used is given based on the algorithm on pages
%      60-61 Solving Least Squares Problems by Lawson and Hanson.
%      And on Fast Plane Rotation Algorithms with Dynamic Scaling
%      by A.A. Anda and H. Park.
%

if  b == 0

   case=1;

   beta=0.0;
   alpha=0.0;
   r=a;
else
   if  d(1)>=d(2)
       flow=1;
       alpha=b/a;
       gamma=d(1)/d(2);
       if  alpha^2 <= gamma

           case=2;

           beta=alpha/gamma;
           delta=1+beta*alpha;
           alpha=-alpha/delta;
           r=a*delta;
           d(1)=d(1)/delta;
           d(2)=d(2)*delta;

        else

           case=3;

           alpha=1/alpha;
           beta=alpha*gamma;
           delta=1+beta*alpha;
           beta=-beta/delta;
           r=b;
```

```
            d1old=d(1);
            d(1)=d(2)*delta;
            d(2)=d1old/delta;
         end
      else
         flow=2;
         alpha=b/a;
         gamma=d(1)/d(2);
         if  alpha^2 <= gamma

            case=2;

            beta=alpha/gamma;
            delta=1+beta*alpha;
            alpha=-alpha;
            beta=beta/delta;
            r=a;
            d(1)=d(1)*delta;
            d(2)=d(2)/delta;

         else

            case=3;

            alpha=1/alpha;
            beta=gamma*alpha;
            delta=1+beta*alpha;
            alpha=alpha/delta;
            r=b*delta;
            d1old=d(1);
            d(1)=d(2)/delta;
            d(2)=d1old*delta;
         end
      end
end
```

A.3.3 Matlab function f2colrot.m

```
function A = f2colrot(A,alpha,beta,p,q,case,flow)
% A = f2colrot(A,alpha,beta,p,q,case,flow)
%
```

```
% This subroutine performs a fast scaled Givens rotation which
% overwrites A with A*F(c,s).
%
% input
%    A          the matrix undergoing rotation (in place)
%    alpha      the q'th row saxpy factor
%    beta       the p'th row saxpy factor
%    p,q        the top and bottom row indices for the rotation
%    case       flag indicating the angle based rotation type
%    flow       flag indicating the diagonal scaling direction
% output
%    A          the matrix rotated in place
%
if case >= 2
    if flow == 1
        if case == 2
            A(:,p)=A(:,p)+beta*A(:,q);
            A(:,q)=A(:,q)+alpha*A(:,p);
        else
            tmp=alpha*A(:,q)-A(:,p);
            A(:,p)=A(:,q)+beta*tmp;
            A(:,q)=tmp;
        end
    else
        if case == 2
            A(:,q)=A(:,q)+alpha*A(:,p);
            A(:,p)=A(:,p)+beta*A(:,q);
        else
            tmp=A(:,q)+beta*A(:,p);
            A(:,q)=alpha*tmp-A(:,p);
            A(:,p)=tmp;
        end
    end
end
```

A.3.4 Matlab function **f2rowrot.m**

```
function A = f2rowrot(A,alpha,beta,p,q,case,flow)
% A = f2rowrot(A,alpha,beta,p,q,case,flow)
%
% This subroutine performs a fast scaled Givens rotation which
```

```
% overwrites A with F(c,s)^T A.
%
% input
%    A          the matrix undergoing rotation (in place)
%    alpha      the q'th row saxpy factor
%    beta       the p'th row saxpy factor
%    p,q        the top and bottom row indices for the rotation
%    case       flag indicating the angle based rotation type
%    flow       flag indicating the diagonal scaling direction
% output
%    A          the matrix rotated in place
%
if case >= 2
    if flow == 1
        if case == 2
            A(p,:)=A(p,:)+beta*A(q,:);
            A(q,:)=A(q,:)+alpha*A(p,:);
        else
            tmp=alpha*A(q,:)-A(p,:);
            A(p,:)=A(q,:)+beta*tmp;
            A(q,:)=tmp;
        end
    else
        if case == 2
            A(q,:)=A(q,:)+alpha*A(p,:);
            A(p,:)=A(p,:)+beta*A(q,:);
        else
            tmp=A(q,:)+beta*A(p,:);
            A(q,:)=alpha*tmp-A(p,:);
            A(p,:)=tmp;
        end
    end
end
```

A.4 Backsolve, common to all factorization algorithms

A.4.1 Matlab function bckslv.m

```
function x = bckslv(A,b)
%
% This subroutine performs backsubstitution to solve a system of
% equations.
%
% input
%    A          upper triangular matrix
%    b          right hand side vector
% output
%    b          the solution vector
%
% (From Algorithm 3.1.2 on page 88 of "Matrix Computations" by Golub
% and Van Loan, 2nd ed.)
%
[m,n]=size(A);
b(n)=b(n)/A(n,n);
for i=n-1:-1:1
    b(i)=(b(i)-A(i,i+1:n)*b(i+1:n))/A(i,i);
end
x=b;
```

Appendix II

JACOBI SYMMETRIC EVD ALGORITHMS AND EXAMPLE

B.1 A Matlab Library of Jacobi Symmetric EVD Algorithms

B.1.1 Matlab function jacobi2slow.m

```
function [h,v,sweep] = jacobi2slow(h,tol)
%
% this function implements algorithm 3.1 from Demmel with the slow
% Jacobi rotation.
%
[n,nn]=size(h);
v=eye(n,n);
sweep=0;
conv=0;
while (conv  ==  0)
% repeat for all pairs of i<j
    for i=1:n-1
        for j=i+1:n
% form the  2x2 submatrix
            a=h(i,i);
            b=h(j,j);
            c=h(i,j);
% compute the Jacobi rotation which will diagonalize the 2x2 matrix
            zeta = (b-a)/(2*c);
            if  zeta == 0
                signz=1;
            else
                signz=sign(zeta);
            end
            t=signz/(abs(zeta)+sqrt(1+zeta*zeta));
            cs=1/sqrt(1+t*t);
            sn=cs*t;
```

```
% update the 2x2 submatrix
            h(i,i)=a-c*t;
            h(j,j)=b+c*t;
            h(i,j)=0;
            h(j,i)=0;
% update the rest of rows and columns i and j
            for k=1:n
                if (k~=i)&(k~=j)
                    tmp=h(i,k);
                    h(i,k)=cs*tmp-sn*h(j,k);
                    h(j,k)=sn*tmp+cs*h(j,k);
                    h(k,i)=h(i,k);
                    h(k,j)=h(j,k);
                end
            end
% update the eigenvetor matrix v
                temp=v(:,i);
                v(:,i)=cs*temp-sn*v(:,j);
                v(:,j)=sn*temp+cs*v(:,j);
        end
    end
% check for convergence
    sweep=sweep+1;
    conv=1;
    for i=1:n-1
        for j=i+1:n
            if abs(h(i,j))/sqrt(h(i,i)*h(j,j)) > tol
                conv=0;
            end
        end
    end
end
```

B.1.2 Matlab function **jacobi2fast.m**

```
function [h,v,sweep,d] = jacobi2fast(h,tol)
%
% this function implements algorithm 3.1 from Demmel with the fast
% Jacobi rotation.
%
[n,nn]=size(h);
v=eye(n,n);
d=v;
sweep=0;
conv=0;
while (conv  ==  0)
% repeat for all pairs of i<j
    for i=1:n-1
        for j=i+1:n
% form the  2x2 submatrix
            a=h(i,i);
            b=h(j,j);
            c=h(i,j)*d(i,i)*d(j,j);
% compute the Jacobi rotation which will diagonalize the 2x2 matrix
            zeta = (b-a)/(2*c);
            if  zeta == 0
                signz=1;
            else
                signz=sign(zeta);
            end
            t=signz/(abs(zeta)+sqrt(1+zeta*zeta));
            cs=1/sqrt(1+t*t);
            sn=cs*t;
            dbeta=d(j,j)/d(i,i);
            dalpha=1/dbeta;
            beta=t*dbeta;
            alpha=t*dalpha;
% update the diagonal factor d
            d(i,i)=cs*d(i,i);
            d(j,j)=cs*d(j,j);
% update the 2x2 submatrix
            h(i,i)=a-c*t;
            h(j,j)=b+c*t;
            h(i,j)=0;
            h(j,i)=0;
```

```
% update the rest of rows and columns i and j
            for k=1:n
                if  (k~=i)&(k~=j)
                    tmp=h(i,k);
                    h(i,k)=tmp-beta*h(j,k);
                    h(j,k)=alpha*tmp+h(j,k);
                    h(k,i)=h(i,k);
                    h(k,j)=h(j,k);
                end
            end
% update the eigenvetor matrix v
            temp=v(:,i);
            v(:,i)=temp-beta*v(:,j);
            v(:,j)=alpha*temp+v(:,j);
        end
    end
% check for convergence
    sweep=sweep+1;
    conv=1;
    for i=1:n-1
        for j=i+1:n
            if  abs(h(i,j)*d(i,i)*d(j,j))/sqrt(h(i,i)*h(j,j)) > tol
                conv=0;
            end
        end
    end
end
for i=1:n-1
    v(i,i)=v(i,i)*d(i,i);
    for j=i+1:n
        df=d(i,i)*d(j,j);
        h(i,j)=h(i,j)*df;
        h(j,i)=h(j,i)*df;
        v(i,j)=v(i,j)*d(j,j);
        v(j,i)=v(j,i)*d(i,i);
    end
end
v(n,n)=v(n,n)*d(n,n);
```

B.1.3 Matlab function jacobi2new.m

```
function [h,v,sweep,d] = jacobi2new(h,tol)
%
% this function implements algorithm 3.1 from Demmel with the new fast
% Jacobi rotation.
%
[n,nn]=size(h);
v=eye(n,n);
d=v;
sweep=0;
conv=0;
while (conv  ==  0)
% repeat for all pairs of i<j
    for i=1:n-1
        for j=i+1:n
% form the  2x2 submatrix
            a=h(i,i);
            b=h(j,j);
            c=h(i,j)*d(i,i)*d(j,j);
% compute the Jacobi rotation which will diagonalize the 2x2 matrix
            zeta = (b-a)/(2*c);
            if  zeta == 0
                signz=1;
            else
                signz=sign(zeta);
            end
            t=signz/(abs(zeta)+sqrt(1+zeta*zeta));
            cs=1/sqrt(1+t*t);
            sn=cs*t;
            dbeta=d(j,j)/d(i,i);
            dalpha=1/dbeta;
            beta=t*dbeta;
            alpha=t*dalpha;
% update the diagonal factor d
            if  d(i,i) >= d(j,j)
                rflag=0;
                d(i,i)=cs*d(i,i);
                d(j,j)=d(j,j)/cs;
                alpha=alpha*cs^2;
            else
                rflag=1;
```

```
                d(i,i)=d(i,i)/cs;
                d(j,j)=cs*d(j,j);
                beta=beta*cs^2;
            end
% update the 2x2 submatrix
            h(i,i)=a-c*t;
            h(j,j)=b+c*t;
            h(i,j)=0;
            h(j,i)=0;
% update the rest of rows and columns i and j
            if rflag == 0
                for k=1:n
                    if (k~=i)&(k~=j)
                        h(i,k)=h(i,k)-beta*h(j,k);
                        h(j,k)=alpha*h(i,k)+h(j,k);
                        h(k,i)=h(i,k);
                        h(k,j)=h(j,k);
                    end
                end
            else
                for k=1:n
                    if (k~=i)&(k~=j)
                        h(j,k)=alpha*h(i,k)+h(j,k);
                        h(i,k)=h(i,k)-beta*h(j,k);
                        h(k,i)=h(i,k);
                        h(k,j)=h(j,k);
                    end
                end
            end
% update the eigenvetor matrix v
            if rflag == 0
                v(:,i)=v(:,i)-beta*v(:,j);
                v(:,j)=alpha*v(:,i)+v(:,j);
            else
                v(:,j)=alpha*v(:,i)+v(:,j);
                v(:,i)=v(:,i)-beta*v(:,j);
            end
        end
    end
% check for convergence
    sweep=sweep+1;
    conv=1;
```

```
    for i=1:n-1
        for j=i+1:n
            if abs(h(i,j)*d(i,i)*d(j,j))/sqrt(h(i,i)*h(j,j)) > tol
                conv=0;
            end
        end
    end
end
for i=1:n-1
    v(i,i)=v(i,i)*d(i,i);
    for j=i+1:n
        df=d(i,i)*d(j,j);
        h(i,j)=h(i,j)*df;
        h(j,i)=h(j,i)*df;
        v(i,j)=v(i,j)*d(j,j);
        v(j,i)=v(j,i)*d(i,i);
    end
end
v(n,n)=v(n,n)*d(n,n);
```

B.2 Jacobi Symmetric EVD Example in Matlab

```
>> tol
tol = 1.000000000000000e-14
>> A
A = 1.000000000000000e+00     1.000000000000000e-01     1.000000000000000e-01
    1.000000000000000e-01     1.000000000000000e+00     1.000000000000000e-01
    1.000000000000000e-01     1.000000000000000e-01     1.000000000000000e+00
>> D
D = 1.000000000000000e+20                          0                          0
                         0     1.000000000000000e+10                          0
                         0                          0     1.000000000000000e+00
>> H=D*A*D
H = 1.000000000000000e+40     9.999999999999999e+28     1.000000000000000e+19
    9.999999999999999e+28     1.000000000000000e+20     1.000000000000000e+09
    1.000000000000000e+19     1.000000000000000e+09     1.000000000000000e+00

>> [Hs,Vs,sweep]=Jacobi2slow(H,tol)
Hs = 1.000000000000000e+40     8.999999999999999e-13    -8.181818181818180e-24
     8.999999999999999e-13     9.900000000000000e+19                         0
    -8.181818181818180e-24                         0     9.818181818181818e-01

Vs = 1.000000000000000e+00    -9.999999999999998e-12    -9.090909090909091e-22
     9.999999999999998e-12     1.000000000000000e+00    -9.090909090909090e-12
     9.999999999999999e-22     9.090909090909090e-12     1.000000000000000e+00

>> [Hf,Vf,sweep,Df]=Jacobi2fast(H,tol)
Hf = 1.000000000000000e+40     8.999999999999999e-13    -8.181818181818180e-24
     8.999999999999999e-13     9.900000000000000e+19                         0
    -8.181818181818180e-24                         0     9.818181818181818e-01
Vf =
     1.000000000000000e+00     9.999999999999998e-12     9.999999999999999e-22
     9.999999999999998e-12     1.000000000000000e+00     9.090909090909090e-12
     9.999999999999999e-22     9.090909090909090e-12     1.000000000000000e+00
Df = 1     0     0
     0     1     0
     0     0     1
>> [Hn,Vn,sweep,Dn]=Jacobi2new(H,tol)
Hn = 1.000000000000000e+40     8.999999999999999e-13    -8.181818181818180e-24
     8.999999999999999e-13     9.900000000000000e+19                         0
    -8.181818181818180e-24                         0     9.818181818181818e-01
Vn =
     1.000000000000000e+00     9.999999999999998e-12     9.999999999999999e-22
     9.999999999999998e-12     1.000000000000000e+00     9.090909090909090e-12
     9.999999999999999e-22     9.090909090909090e-12     1.000000000000000e+00
Dn = 1     0     0
     0     1     0
     0     0     1
```

```
>> [Xeig,Deig]=eig(H)
Xeig = 9.999954029260542e-12     1.000000000000000e+00                          0
    -1.000000000000000e+00     9.999954029260535e-12    -4.667747987570129e-15
     4.667747987570128e-15                         0    -1.000000000000000e+00
Deig =
    -1.928125821797580e+23                         0                          0
                         0     1.000000000000000e+40                          0
                         0                         0                          0
```