# *Topics in Matrix Computations: Stability and Efficiency of Algorithms*

Hargreaves, Gareth

2005

MIMS EPrint: **2006.357**

Manchester Institute for Mathematical Sciences
School of Mathematics

The University of Manchester

Reports available from: http://eprints.maths.manchester.ac.uk/
And by contacting: The MIMS Secretary
School of Mathematics
The University of Manchester
Manchester, M13 9PL, UK

ISSN 1749-9097

# TOPICS IN MATRIX COMPUTATIONS: STABILITY AND EFFICIENCY OF ALGORITHMS

A thesis submitted to The University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2005

**Gareth I. Hargreaves**
School of Mathematics

# Contents

# List of Tables

# List of Figures

# Abstract

Numerical algorithms are considered for three distinct areas of numerical linear algebra: hyperbolic matrix computations, condition numbers of structured matrices, and trigonometric matrix functions.

We first consider hyperbolic rotations and show how to construct them accurately. A new accurate representation is devised which also avoids overflow. We show how to apply hyperbolic rotations directly, in mixed form, and by the OD procedure, with a rounding error analysis that shows the latter two methods are stable. A rounding error analysis for combining a sequence of nonoverlapping hyperbolic rotations applied in mixed form or by the OD procedure is then given. Applying a hyperbolic rotation directly is generally thought to be unstable but no proof has previously been given. However, using numerical experiments we prove that it is unstable. We describe several methods of applying fast hyperbolic rotations and unified rotations, giving a rounding error analysis and numerical experiments to show which are stable and which are not. Hyperbolic Householder transformations are briefly discussed.

We then consider the hyperbolic QR factorization for which we present new results for the existence of the closely related HR factorization, and then use these to prove new theorems for the existence of the hyperbolic QR factorization. We describe how nonoverlapping hyperbolic rotations can be used to compute the hyperbolic QR factorization, with a rounding error analysis to show that this method is stable. Two applications of the hyperbolic QR

factorization are also discussed.

For an $n \times n$ tridiagonal matrix we exploit the structure of its QR factorization to devise two new algorithms for computing the 1-norm condition number in $O(n)$ operations. The algorithms avoid underflow and overflow, and are simpler than existing algorithms since tests are not required for degenerate cases. An error analysis of the first algorithm is given, while the second algorithm is shown to be competitive in speed with existing algorithms. We then turn our attention to an $n \times n$ diagonal-plus-semiseparable matrix, $A$, for which several algorithms have recently been developed to solve $Ax = b$ in $O(n)$ operations. We again exploit the QR factorization of the matrix to present an algorithm that computes the 1-norm condition number in $O(n)$ operations.

We also consider algorithms for computing the matrix cosine. The algorithms scale a matrix by a power of two to make the norm of the scaled matrix small, use a Padé approximation to compute the cosine of the scaled matrix, and recover the cosine of the original matrix using the double angle formula $\cos(2A) = 2\cos^2(A) - I$. We make several improvements to an algorithm of Higham and Smith to derive new algorithms, which are shown by theory and numerical experiments to bring increased efficiency and accuracy. We also consider an algorithm for simultaneously computing $\cos(A)$ and $\sin(A)$ that extends the ideas for the cosine and intertwines the cosine and sine double angle recurrences.

# Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

# Copyright

# Statement

- The material in Chapter 3 is based on the technical report "Computing the Condition Number of Tridiagonal and Diagonal-Plus-Semiseparable Matrices in Linear Time," Numerical Analysis Report No. 447, Manchester Centre for Computational Mathematics, April 2004. This work is to appear in the SIAM Journal on Matrix Analysis and Applications.

- The material in Chapter 4 is based on the technical report "Efficient Algorithms for the Matrix Cosine and Sine" (with Nicholas J. Higham), Numerical Analysis Report No. 461, Manchester Centre for Computational Mathematics, February 2005. This work is to appear in Numerical Algorithms.

# Acknowledgements

I am extremely grateful to my supervisor, Nick Higham, for sharing his knowledge and expertise, and providing excellent guidance in both the research and writing of this thesis.

I would like to thank my fellow students Michael Berhanu, Craig Lucas and Anna Mills for their help and advise. Thank you also to Sven Hammarling and Françoise Tisseur for their useful suggestions.

Both my immediate family and parents-in-law, have offered much encouragement and support over the past three years for which I am very grateful.

Finally, a special thank you goes to my wife, Jenni, for her constant love, encouragement and patience.

# Chapter 1

# Introduction

## 1.1 Basic Definitions

Throughout the thesis we will require the definition of the following types of matrices.

- $A \in \mathbb{R}^{n\times n}$ is *symmetric* if $A = A^T$, where $A^T$ denotes the transpose of $A$. Also, $A \in \mathbb{C}^{n\times n}$ is *Hermitian* if $A = A^*$ where $A^*$ denotes the conjugate transpose of $A$.

- $A \in \mathbb{C}^{n\times n}$ is *diagonal* if $a_{ij} = 0$ for $i \neq j$. We use the notation $A = \mathrm{diag}(x_1, x_2, \ldots, x_n)$ for a diagonal matrix with $a_{ii} = x_i$ for $i = 1\colon n$.

- $A \in \mathbb{C}^{n\times n}$ is *upper (lower) triangular* if $a_{ij} = 0$ for $i > j$ $(i < j)$. We also say that $A$ is *strictly upper (lower) triangular* if it is upper (lower) triangular with $a_{ii} = 0$ for $i = 1\colon n$.

- $A \in \mathbb{C}^{n\times n}$ is *upper (lower) bidiagonal* if $a_{ij} = 0$ for $i > j$ $(j > i)$ and $i + 1 < j$ $(j + 1 < i)$.

- $A \in \mathbb{C}^{n\times n}$ is *tridiagonal* if $a_{ij} = 0$ for $i + 1 < j$ and $j + 1 < i$.

- $A \in \mathbb{C}^{n\times n}$ is *upper (lower) Hessenberg* if $a_{ij} = 0$ for $i > j + 1$ $(i < j - 1)$.

- $A \in \mathbb{C}^{n\times n}$ is *positive definite* if $x^*Ax > 0$ for all $x \in \mathbb{C}^n$, $x \neq 0$.

- $A \in \mathbb{R}^{n\times n}$ is *orthogonal* if $A^TA = I$ where $I$ is the $n \times n$ *identity matrix* with ones on the diagonal and zeros elsewhere. We also say that $A \in \mathbb{C}^{n\times n}$ is *unitary* if $A^*A = I$.

The following linear algebra definitions will also be used:

- The set of vectors $S = \{a_1, a_2, \ldots, a_n\} \subset \mathbb{R}^m$ is *linearly independent* if the only solution to
$$\sum_{i=1}^{n} \alpha_i a_i = 0, \quad \alpha_i \in \mathbb{R},$$
is $\alpha_i = 0$ for all $i$.

- The *span* of a set of vectors is the set of all linear combinations of the vectors. Therefore $\text{span}(S)$ is given by
$$\text{span}\{a_1, a_2, \ldots, a_n\} = \left\{ \sum_{i=1}^{n} \beta_i a_i : \beta_i \in \mathbb{R} \right\}.$$

- The *null space* of $A \in \mathbb{R}^{n\times n}$ is given by
$$\text{null}(A) = \{x \in \mathbb{R}^n : Ax = 0\}.$$

- Let $A = [a_1\ a_2\ \ldots\ a_n]$ be a column partitioning. Then the *range* of $A$ is defined by
$$\text{ran}(A) = \text{span}\{a_1, a_2, \ldots, a_n\}.$$

- The *rank* of $A$ is defined by
$$\text{rank}(A) = \dim(\text{ran}(A)),$$
which is the maximum number of linearly independent vectors of $\text{ran}(A)$.

- If $A \in \mathbb{R}^{n\times n}$ and $\text{rank}(A) = n$, then $A$ is said to be *nonsingular* or *full rank*, and there exists an *inverse* of $A$, denoted $A^{-1}$ such that $AA^{-1} = I$. If $\text{rank}(A) \neq n$ we say that $A$ is *singular* or *rank deficient* and no inverse exists. If $A \in \mathbb{R}^{m\times n}$ and $\text{rank}(A) = \min(m, n)$ then $A$ is said to have *full rank*. Otherwise it is said to be *rank deficient*.

## 1.2 Norms and Condition Numbers

Norms are a valuable tool which provides a measure of size for vectors and matrices. Extensive use will be made of norms and their properties, particularly for conducting rounding error analysis and bounding perturbations.

### 1.2.1 Vector Norms

A *vector norm* on $\mathbb{C}^n$ is a function $\|\cdot\| : \mathbb{C}^n \to \mathbb{R}$ that has the following properties.

1. $\|x\| \geq 0$ and $\|x\| = 0$ if and only if $x = 0$.
2. $\|\alpha x\| = |\alpha|\|x\|$ for all $\alpha \in \mathbb{C}, x \in \mathbb{C}^n$.
3. $\|x + y\| \leq \|x\| + \|y\|$ for all $x, y \in \mathbb{C}^n$.

A valuable class of vector norms are the Hölder $p$-norms, defined by

$$\|x\|_p = \left( \sum_{i=1}^{n} |x_i|^p \right)^{1/p}, \quad p \geq 1. \tag{1.1}$$

The most commonly used of the $p$-norms are the 1, 2 and $\infty$ norms:

$$\begin{aligned} \|x\|_1 &= \sum_{i=1}^{n} |x_i|, \\ \|x\|_2 &= \left( \sum_{i=1}^{n} |x_i|^2 \right)^{1/2}, \\ \|x\|_\infty &= \max_{1\leq i\leq n} |x_i|. \end{aligned}$$

For unitary $Q$ we have $Q^*Q = I$ and so

$$\|Qx\|_2^2 = x^*Q^*Qx = x^*x = \|x\|_2^2.$$

We therefore say that the 2-norm is invariant under unitary transformations.

### 1.2.2 Matrix Norms

Analogous to the vector norm we have the *matrix norm* $\|\cdot\| : \mathbb{C}^{m\times n} \to \mathbb{R}$ that has the following properties.

1. $\|A\| \geq 0$ and $\|A\| = 0$ if and only if $A = 0$.
2. $\|\alpha A\| = |\alpha|\|A\|$ for all $\alpha \in \mathbb{C}, A \in \mathbb{C}^{m\times n}$.
3. $\|A + B\| \leq \|A\| + \|B\|$ for all $A, B \in \mathbb{C}^{m\times n}$.

Among the most important matrix norms is the Frobenius norm,

$$\|A\|_F = \left(\sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2\right)^{1/2}$$

and the subordinate $p$-norms,

$$\|A\|_p = \max_{x\neq 0} \frac{\|Ax\|_p}{\|x\|_p},$$

which are defined in terms of the $p$-norms given by (1.1). It can be shown that the 1, 2 and $\infty$ matrix $p$-norms satisfy

$$\|A\|_1 = \max_{1\leq j\leq n} \sum_{i=1}^{m} |a_{ij}|,$$
$$\|A\|_\infty = \max_{1\leq i\leq m} \sum_{j=1}^{n} |a_{ij}|,$$
$$\|A\|_2 = (\rho(A^*A))^{1/2} = \sigma_{\max}(A),$$

where the spectral radius

$$\rho(B) = \max(|\lambda| : \det(B - \lambda I) = 0)$$

and $\sigma_{\max}(A)$ denotes the largest singular value of $A$.

The Frobenius norm and matrix $p$-norms are said to be *consistent* since they satisfy the useful property

$$\|AB\| \leq \|A\|\|B\|.$$

The 2-norm and Frobenius norm are both *unitarily invariant*, which means that for any unitary matrices $U$ and $V$ we have $\|UAV\| = \|A\|$.

We will make use of the following lemmas.

**Lemma 1.2.1.** *Let $A, B \in \mathbb{R}^{n\times n}$. If $|A| \leq |B|$ then $\|A\|_F \leq \|B\|_F$.*

**Lemma 1.2.2 ([30, Sec. 6.2]).** *Let $A \in \mathbb{R}^{n\times n}$. Then $1/\sqrt{n}\|A\|_F \leq \|A\|_1 \leq \sqrt{n}\|A\|_F$.*

Another important quantity in numerical linear algebra is the condition number as it provides a measure of the sensitivity of a problem to perturbations in the data. The condition number with respect to matrix inversion of a nonsingular $A \in \mathbb{C}^{n\times n}$ is defined by

$$\kappa_p(A) := \lim_{\epsilon\to 0} \sup_{\|\Delta A\|_p \leq \epsilon\|A\|_p} \left( \frac{\|(A+\Delta A)^{-1} - A^{-1}\|_p}{\epsilon\|A^{-1}\|_p} \right),$$

where $\|\cdot\|$ denotes a subordinate matrix $p$-norm. An explicit formula for this condition number (see [30, Thm. 6.4]) is given by

$$\kappa_p = \|A\|_p\|A^{-1}\|_p.$$

## 1.3 Floating Point Numbers

A floating point number system $F \subset \mathbb{R}$ is a subset of the real numbers and is defined by the *base* $\beta$, *precision* $t$ and *exponent range* $e_{min} \leq e \leq e_{max}$. The elements of $F$ have the form

$$y = \pm m \times \beta^{e-t},$$

where the *significand* $m$ is an integer satisfying $0 \le m \le \beta^{t-1}$.

The element of $F$ nearest to $x \in \mathbb{R}$ is denoted by $fl(x)$. A bound for the error in the approximation $fl(x)$ to $x$ is given in the following theorem.

**Theorem 1.3.1 ([30, Thm. 2.2]).** *If $x \in \mathbb{R}$ lies in the range of $F$ then*

$$fl(x) = x(1+\delta), \quad |\delta| < u = \frac{1}{2}\beta^{1-t}. \tag{1.2}$$

We call $u$ in (1.2) the *unit roundoff*.

An operation involving floating point numbers, such as addition, subtraction, multiplication and division, is called a *floating point operation* or *flop*. We will also use this notation for the square root of a floating point number.

We say that $fl(x)$ *overflows* if $|fl(x)| > \max\{|y|: y \in F\}$ and *underflows* if $0 < |fl(x)| < \min\{|y|: 0 \ne y \in F\}$. Clearly, underflow or overflow is undesirable and should be avoided where possible.

## 1.4 Model of Floating Point Arithmetic

Throughout this thesis we denote quantities evaluated in floating point arithmetic with a hat or by the $fl(\cdot)$ notation. We will use the standard model for floating point arithmetic:

$$\begin{aligned} fl(x \text{ op } y) &= (x \text{ op } y)(1+\delta_1), \quad |\delta_1| \le u, \quad \text{op} = +, -, /, *, \\ fl(\sqrt{x}) &= \sqrt{x}(1+\delta_2), \quad |\delta_2| \le u, \end{aligned} \tag{1.3}$$

where $u$ is the unit roundoff of the computer. The following modification of (1.3) will also be used:

$$fl(x \text{ op } y) = \frac{x \text{ op } y}{1+\delta_3}, \quad |\delta_3| \le u.$$

The following lemmas are used without comment.

**Lemma 1.4.1 ([30, Lem. 3.1]).** *If $|\delta_i| \le u$ and $\rho_i = \pm 1$ for $i = 1\colon n$, and $nu < 1$, then*

$$\prod_{i=1}^{n}(1+\delta_i)^{\rho_i} = 1 + \theta_n,$$

*where*

$$|\theta_n| \le \frac{nu}{1-nu} =: \gamma_n.$$

**Lemma 1.4.2 ([30, Lem. 3.3]).** *For any positive integer $k$ let $\theta_k$ denote a quantity bounded according to $|\theta_k| \le \gamma_k = ku/(1-ku)$. The following relations hold:*

$$\begin{aligned}
(1+\theta_k)(1+\theta_j) &= 1+\theta_{k+j},\\
\frac{1+\theta_k}{1+\theta_j} &= \begin{cases} 1+\theta_{k+j}, & j \le k,\\ 1+\theta_{k+2j}, & j > k,\end{cases}\\
\gamma_k\gamma_j &\le \gamma_{\min(k,j)} \quad \textit{for } \max(j,k)u \le 1/2,\\
i\gamma_k &\le \gamma_{ik},\\
\gamma_k + u &\le \gamma_{k+1},\\
\gamma_k + \gamma_j + \gamma_k\gamma_j &\le \gamma_{k+j}.
\end{aligned} \tag{1.4}$$

**Lemma 1.4.3.** *For any positive integer $k$ let $\gamma_k = ku/(1-ku)$. Then*

$$\frac{1+\gamma_k}{1+\gamma_j} \le \begin{cases} 1+\gamma_{k+j}, & j \le k,\\ 1+\gamma_{k+2j}, & j > k.\end{cases}$$

*Proof.* The proof is almost identical to the proof of (1.4) as in [30, Lem. 3.3]. □

We also make use of the constant

$$\widetilde{\gamma}_n = \frac{cnu}{1-cnu},$$

where $c$ denotes a small integer constant whose exact value is unimportant.

## 1.5 Forward and Backward Errors

Using the properties in Section 1.4 it is often possible to conduct a rounding error analysis of a numerical algorithm to obtain backward and forward error results. These results provide a measure of the accuracy of a solution computed using a numerical algorithm. In order to define what we mean by forward and backward errors, and also the stability of an algorithm, we denote by $\widehat{y}$ the vector $y = f(x)$ computed in floating point arithmetic, where $f$ is a vector function and $x$ is a vector. We consider only this vector problem for simplicity but the ideas extend to other problems.

- A backward error analysis involves bounding $|\Delta x|$ where $\Delta x$ satisfies $\widehat{y} = f(x + \Delta x)$. We say a method is *backward stable* if it produces a computed $\widehat{y}$ such that $\widehat{y} = f(x + \Delta x)$ for some small $\Delta x$, where the definition of "small" depends on the problem.

- A forward error analysis involves bounding the absolute error $|\widehat{y} - y|$. A method is said to be *forward stable* if the bound is small, where the definition of "small" depends on the problem.

- A mixed forward-backward error analysis involves bounding $|\Delta x|$ and $|\Delta y|$ where $\Delta x$ and $\Delta y$ satisfy $\widehat{y} + \Delta y = f(x + \Delta x)$. A method is said to be *mixed forward-backward stable* if $|\Delta x|$ and $|\Delta y|$ are small, where the definition of "small" depends on the problem.

In general we call an algorithm *stable* if it is mixed forward-backward stable. Therefore a backward stable algorithm is said to be stable.

The forward and backward errors for a problem are connected by the conditioning of the problem, that is, the sensitivity of the solution to perturbations in the data. When the backward error, forward error, and the condition number

are defined in a consistent fashion we have the useful rule of thumb

$$\text{forward error} \lesssim \text{condition number} \times \text{backward error},$$

with approximate equality possible.

## 1.6 Orthogonal Transformations and Factorizations

In this section we describe two commonly used orthogonal transformations that introduce zeros into vectors and will be required in later chapters. We also describe the QR factorization, which may be computed using the transformations described.

### 1.6.1 Givens Rotations

A *Givens rotation*, $Q \in \mathbb{C}^{n\times n}$ is a unitary transformation that is equal to the identity matrix except that

$$Q([i,j],[i,j]) = \begin{bmatrix} \bar{c} & \bar{s} \\ -s & c \end{bmatrix},$$

where $|c|^2 + |s|^2 = 1$, with $|c| = \cos\theta$ and $|s| = \sin\theta$ for some $\theta$. Applying a Givens rotation to a matrix from the left affects only two rows and applying it from the right affects two columns. For this reason, applying a Givens rotation is not implemented as full matrix multiplication. For convenience we consider a rotation to be a $2 \times 2$ matrix which contains the significant components of the rotation:

$$G = \begin{bmatrix} \bar{c} & \bar{s} \\ -s & c \end{bmatrix}.$$

Givens rotations are used to introduce zeros into a vector and hence reduce matrices to particular forms. Given a vector $x \in \mathbb{C}^2$, $c$ and $s$ can be chosen so

that $y = Gx$ has zero second component. If the Givens rotation is applied to $x \in \mathbb{C}^2$ then

$$y_1 = \bar{c}x_1 + \bar{s}x_2,$$

$$y_2 = -sx_1 + cx_2,$$

and so $y_2 = 0$ if

$$c = \frac{x_1}{\sqrt{|x_1|^2 + |x_2|^2}} \quad \text{and} \quad s = \frac{x_2}{\sqrt{|x_1|^2 + |x_2|^2}}. \tag{1.5}$$

Thus, the transformation gives

$$Gx = \begin{bmatrix} \bar{c} & \bar{s} \\ -s & c \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \sqrt{|x_1|^2 + |x_2|^2} \\ 0 \end{bmatrix}.$$

In practice, $c$ and $s$ are usually rewritten to avoid overflow in floating point arithmetic, so that if $|x_1| > |x_2|$,

$$c = \frac{\text{sign}(x_1)}{\sqrt{1 + |t|^2}}, \quad s = ct, \quad \text{with} \quad t = \frac{x_2}{x_1},$$

and

$$c = st, \quad s = \frac{\text{sign}(x_2)}{\sqrt{1 + |t|^2}}, \quad \text{with} \quad t = \frac{x_1}{x_2},$$

otherwise, where $\text{sign}(x) = x/|x|$ for $x \in \mathbb{C}$. The Givens rotation can be applied to a matrix $A \in \mathbb{C}^{2\times n}$ by forming $B = GA$, and it is well documented that this is a stable process (see [30]). The cost of applying a Givens rotation to $A \in \mathbb{C}^{2\times n}$ is $6n$ operations.

### 1.6.2 Householder Transformations

A *Householder reflector* is a unitary matrix of the form

$$Q = I - \beta vv^*, \quad 0 \neq v \in \mathbb{C}^n,$$

with $\beta$ on the circle $|\beta - r| = |r|$, where $r = -1/(v^*v)$. For any distinct vectors $x \in \mathbb{C}^n$ and $y \in \mathbb{C}^n$ such that $x^*x = y^*y$, we can choose $v = y - x$ and

$\beta = 1/(v^*x)$ so that $Qx = y$. This allows us to apply a Householder reflector in order to zero all but one element of a vector $x \in \mathbb{C}^n$, which is commonly achieved by $y = \text{sign}(x_i)\sqrt{x^*x}e_i$, where $e_i$ is the $i$th column of the identity matrix. This choice results in a Hermitian matrix $Q$. An alternative choice used in LAPACK is $y = \pm\sqrt{x^*x}e_i$. This sends $x_i$ to a real multiple of $e_i$ and is useful for problems where it is advantageous to apply a real algorithm to the resulting vector instead of a complex algorithm. Using this choice results in a non Hermitian matrix $Q$.

When applying a Householder reflector to a matrix it is important to exploit the structure of $Q$ to reduce the number of flops. We note that if $A \in \mathbb{C}^{m\times n}$ then

$$QA = (I - \beta vv^*)A = A - vw^*,$$

where $w = \beta A^*v$. We can therefore apply a Householder reflector to a matrix $A \in \mathbb{C}^{m\times n}$ by a matrix-vector multiplication and the calculation of an outer product, which in total requires $4mn$ operations. If $Q$ is applied to $A$ by forming $Q$ and then multiplying $Q$ and $A$, the number of flops is increased by an order of magnitude.

### 1.6.3 QR Factorization

A *QR factorization* of $A \in \mathbb{C}^{m\times n}$ with $m \geq n$ is

$$A = QR = Q\begin{bmatrix} R_1 \\ 0 \end{bmatrix},$$

where $Q$ is unitary and $R_1 \in \mathbb{C}^{n\times n}$ is upper triangular. If $A$ is real then $Q$ is orthogonal and $R_1$ is also real.

The most common method of computing the QR factorization is by Householder reflectors. The matrix $A \in \mathbb{C}^{m\times n}$ is reduced to upper trapezoidal form by applying a sequence of Householder reflectorss to zero $A^{(k)}(k+1:m,k)$,

where $A^{(k)}$ is the matrix $A$ after $k-1$ Householder reflectors and $A^{(1)} = A$. This gives the upper trapezoidal $R$ as

$$R = H_t \dots H_2 H_1 A = Q^* A,$$

where $H_i$ is the $i$th Householder reflector and $t = \min(m-1, n)$.

## 1.7 Matrix Functions

A *matrix function* can have various meanings, but we will only be concerned with a definition that takes a scalar function, $f$, and defines the equivalent matrix function $f(A)$ to have the same dimensions as $A \in \mathbb{C}^{n \times n}$. When $f(x)$ is a polynomial or rational function, we will regard a matrix function to be the function obtained by replacing a scalar variable $x \in \mathbb{C}$ in the scalar function $f(x)$, by the matrix $A \in \mathbb{C}^{n \times n}$. We replace division by matrix inversion (providing the inverse exists), and replace 1 by the identity matrix. For example, the scalar function

$$f(x) = \frac{2 + x + 3x^2}{1 - x}$$

leads to the matrix function

$$f(A) = (2I + A + 3A^2)(I - A)^{-1}, \quad \text{if } 1 \notin \Lambda(A),$$

where $\Lambda(A)$ denotes the set of eigenvalues of $A$, which is called the *spectrum* of $A$. Similarly scalar functions defined by a power series extend to matrix functions, such as

$$\log(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots, \quad |x| < 1,$$

and

$$\log(1 + A) = A - \frac{A^2}{2} + \frac{A^3}{3} - \frac{A^4}{4} + \cdots, \quad \rho(A) < 1,$$

where $\rho(A) = \max\{|\lambda| : \lambda \in \Lambda(A)\}$ is called the *spectral radius* and the condition $\rho(A) < 1$ ensures convergence of the matrix series. Many power series have infinite radius of convergence such as

$$\cos(A) = I - \frac{A^2}{2!} + \frac{A^4}{4!} - \cdots.$$

For more details on the radius of convergence of Taylor series of matrix functions see [26].

This approach to defining a matrix function is sufficient for a wide range of functions, but it does not provide a definition for a general matrix function. There are many equivalent formal definitions of a matrix function of which we present two here.

The first definition is based on the Jordan canonical form of a matrix $A \in \mathbb{C}^{n\times n}$:

$$Z^{-1}AZ = J = \operatorname{diag}(J_1, J_2, \ldots, J_p), \tag{1.6a}$$

$$J_k = J_k(\lambda_k) = \begin{bmatrix} \lambda_k & 1 & & \\ & \lambda_k & \ddots & \\ & & \ddots & 1 \\ & & & \lambda_k \end{bmatrix} \in \mathbb{C}^{m_k\times m_k}, \tag{1.6b}$$

where $Z$ is nonsingular and $m_1 + m_2 + \cdots + m_p = n$.

We denote by $\lambda_1, \ldots, \lambda_s$ the distinct eigenvalues of $A$ and let $n_i$ be the order of the largest Jordon block in which $\lambda_i$ appears. We also define the values

$$f^{(j)}(\lambda_i), \quad j = 0\colon n_i - 1, \quad i = 1\colon s$$

to be the values of the function $f$ on the spectrum of $A$, and if they exist $f$ is said to be defined on the spectrum of $A$.

**Definition 1.7.1 ([20, Thm 11.1.1]).** *Let $A \in \mathbb{C}^{n\times n}$ have the Jordon canonical form* (1.6) *and $f$ be defined on the spectrum of $A$. Then*

$$f(A) := Zf(J)Z^{-1} = Z\operatorname{diag}(f(J_k))Z^{-1},$$

*where*

$$f(J_k) := \begin{bmatrix} f(\lambda_k) & f'(\lambda_k) & \cdots & \frac{f^{(m_k-1)}(\lambda_k)}{(m_k-1)!} \\ & f(\lambda_k) & \ddots & \vdots \\ & & \ddots & f'(\lambda_k) \\ & & & f(\lambda_k) \end{bmatrix}.$$

Alternatively, given a function $f$ that is analytic inside and on a closed contour $\Gamma$ which encloses the eigenvalues of $A$, $f(A)$ can be defined by a generalisation of the Cauchy integral theorem as

$$f(A) = \frac{1}{2\pi i} \int_\Gamma f(z)(zI - A)^{-1} dz.$$

### 1.7.1 Evaluating Matrix Polynomials

Evaluating a matrix polynomial is a common task when working with matrix functions. Horner's method is almost always used when evaluating a scalar polynomial, but in the matrix case we must also consider alternative methods.

The most obvious method of evaluating the matrix polynomial,

$$p_m(A) = \sum_{i=0}^{m} b_i A^i, \quad A \in \mathbb{C}^{n\times n}, \tag{1.7}$$

is to explicitly form each power of $A$.

**Algorithm 1.7.2.** *Evaluates the matrix polynomial* (1.7) *by explicitly forming matrix powers.*

```
1  A_0 = I, A_1 = A
2  for i = 2: m
3        A_i = A * A_{i-1}
4  end
5  p_m = sum_{i=0}^{m} b_i A_i
```

This method is of particular interest when $m$ is not known or if we wish to evaluate several polynomials in $A$.

An alternative method is Horner's method extended for a matrix polynomial.

**Algorithm 1.7.3.** *Evaluates the matrix polynomial* (1.7) *by Horner's method.*

```
1  S_{m-1} = b_m A + b_{m-1} I
2  for i = m - 2: -1: 0
3        S_i = A S_{i+1} + b_i
4  end
5  p_m = S_0
```

The cost of Algorithms 1.7.2 and 1.7.3 is $(m-1)M$, where $M$ denotes a matrix multiplication.

A more efficient method of evaluating a matrix polynomial is that of Paterson and Stockmeyer [43]. This expresses $p_m(A)$ as

$$p_m(A) = \sum_{k=0}^{r} B_k (A^s)^k, \quad r = \text{floor}(m/s), \tag{1.8}$$

where $s$ is an integer parameter and

$$B_k = \begin{cases} b_{sk+s-1}A^{s-1} + \cdots + b_{sk+1}A + b_{sk}I, & k = 0\colon r-1, \\ b_m A^{m-sr} + \cdots + b_{sr+1}A + b_{sr}I, & k = r. \end{cases}$$

The powers $A^2, \ldots, A^s$ are computed and the $B_k$ evaluated. Finally (1.8) is evaluated using Horner's method. As an example, for $m = 8$ and $s = 3$ we have,

$$p_8(A) = B_0 + B_1 A^3 + B_2 (A^3)^2,$$

where

$$B_0 = b_0 I + b_1 A + b_2 A^2,$$

$$B_1 = b_3 I + b_4 A + b_5 A^2,$$

$$B_2 = b_6 I + b_7 A + b_8 A^2.$$

The total cost of evaluating $p_m(A)$ by the Paterson-Stockmeyer method is

$$(s + r - 1 - f(s,m))M, \quad f(s,m) = \begin{cases} 1 & \text{if } s \text{ divides } m, \\ 0 & \text{otherwise.} \end{cases} \tag{1.9}$$

This quantity is approximately minimised by $s = \sqrt{m}$. Therefore we take $s$ to be one of floor($\sqrt{m}$) and ceil($\sqrt{m}$). In the following theorem we show that both choices of $s$ yield the same cost.

**Theorem 1.7.4.** *For the Paterson-Stockmeyer method of evaluating the matrix polynomial* (1.7)*, the cost given by* (1.9) *is the same for both* $s = \text{ceil}(\sqrt{m})$ *and* $s = \text{floor}(\sqrt{m})$.

*Proof.* Given an integer $m$, then either $m = i^2$ or $i^2 < m < (i+1)^2$, for some integer $i$. If $m = i^2$ then $\text{floor}(\sqrt{m}) = \text{ceil}(\sqrt{m})$ and hence both choices of $s$ give the same cost.

Assume that $i^2 < m < (i+1)^2$. Then $i < \sqrt{m} < i+1$ and hence the two choices of $s$, denoted by $s_1$ and $s_2$ are

$$s_1 = \text{floor}(\sqrt{m}) = i,$$
$$s_2 = \text{ceil}(\sqrt{m}) = i+1.$$

Since $i^2 < m < (i+1)^2$, $s_1$ divides $m$ only if $m = i^2+i$ or $m = i^2+2i$. Similarly $s_2$ divides $m$ only if $m = i^2 + i$. Hence the possible values of $s$ lead to

$$f_1(s_1, m) = \begin{cases} 1 & \text{if } m = i^2 + i \text{ or } m = i^2 + 2i, \\ 0 & \text{otherwise,} \end{cases}$$

$$f_2(s_2, m) = \begin{cases} 1 & \text{if } m = i^2 + i, \\ 0 & \text{otherwise.} \end{cases}$$

If $i^2 < m < i^2 + i$ then $i < m/s_1 < i+1$ and hence $\text{floor}(m/s_1) = i$. Using a similar argument for the other cases we find

$$r_1 = \text{floor}(m/s_1) = \begin{cases} i & \text{if } m < i^2 + i, \\ i+1 & \text{if } i^2 + i \le m < i^2 + 2i, \\ i+2 & \text{if } m = i^2 + 2i, \end{cases}$$

$$r_2 = \text{floor}(m/s_2) = \begin{cases} i-1 & \text{if } m < i^2 + i, \\ i & \text{if } m \ge i^2 + i. \end{cases}$$

Testing the various possible cases we find that $s_1+r_1-1-f_1 = s_2+r_2-1-f_2$, and hence both choices of $s$ give the same cost. □

The advantage of the Paterson-Stockmeyer method over Algorithms 1.7.2 and 1.7.3 is the reduced number of matrix multiplications needed. This is highlighted by Table 1.1 which compares the number of multiplications required by the three methods described to evaluate matrix polynomials of various degrees.

Table 1.1: The total number of matrix multiplications required by the Paterson Stockmeyer method and Algorithms 1.7.2 and 1.7.3 to evaluate the matrix polynomial of degree $m$.

| $m$ | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PS method | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 4 | 5 | 5 | 5 | 6 | 6 |
| Algs 1.7.2/1.7.3 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |

The disadvantage of the Paterson-Stockmeyer method is that $(s+2)n^2$ elements of storage are required. This can be reduced to $4n^2$ using the variation of the Paterson-Stockmeyer method of Van Loan [54], which computes $p_m$ a column at a time but costs approximately 40% more than the original method.

### 1.7.2 Conditioning of Matrix Functions

If a matrix function is computed in floating point arithmetic then it is subject to rounding errors. Through the use of a rounding error analysis, these errors can often be interpreted as perturbations in the data. In order to determine the potential accuracy of a computed matrix function it is important to be able to measure the sensitivity of $f(X)$ to perturbations in $X$. This sensitivity can be measured by the *matrix condition number*

$$\mathrm{cond}(f,X) = \lim_{\delta\to 0} \sup_{\|E\|_F \le \delta\|X\|_F} \frac{\|f(X+E)-f(X)\|_F}{\delta\|f(X)\|_F}. \tag{1.10}$$

A large condition number shows that $f(A)$ is particularly sensitive to perturbations in the data and hence may not be accurate when computed in floating point arithmetic. We now show how the condition number (1.10) can be estimated [37].

The linear operator $L(X, \cdot): \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$ is said to be the *Fréchet derivative* of $f: \mathbb{C}^{n\times n} \to \mathbb{C}^{n\times n}$ at $X \in \mathbb{C}^{n\times n}$ if for all matrices $E \in \mathbb{C}^{n\times n}$

$$f(X+E) - f(X) - L(X,E) = O(\|E\|).$$

We denote by $L(X,E)$ the Fréchet derivative of $f$ at $X$ in the direction $E$. For a matrix function with the power series representation $f(X) = \sum_{i=0}^{\infty} \alpha_i X^i$ we have

$$L(X,E) = \sum_{k=1}^{\infty} \alpha_k \sum_{j=1}^{k} X^{j-1} E X^{k-j}.$$

A relationship between the Fréchet derivative and the condition number (1.10) is given by (see [26])

$$\mathrm{cond}(f,X) = \frac{\|L(X,\cdot)\|_F \|X\|_F}{\|f(X)\|_F},$$

where

$$\|L(X,\cdot)\|_F := \max_{\|E\|_F \neq 0} \frac{\|L(X,E)\|_F}{\|E\|_F}.$$

Hence, an estimate of $\|L(X,\cdot)\|_F$ will lead to an estimate of $\mathrm{cond}(f,X)$.

The Kronecker form of the Fréchet derivative is said to be given by $K(X) \in \mathbb{C}^{n^2\times n^2}$ if it satisfies

$$\mathrm{vec}(L(X,E)) = K(X)\mathrm{vec}(E), \tag{1.11}$$

where $\mathrm{vec}(A) = [a_1^T \ a_2^T \ \ldots \ a_n^T]^T$ for a matrix with column partition $A = [a_1 \ a_2 \ \ldots \ a_n]$. If the matrix function has the power series representation $f(X) = \sum_{i=0}^{\infty} \alpha_i X^i$ then the Kronecker form of the Fréchet derivative is

$$K(X) = \sum_{k=1}^{\infty} \alpha_i \sum_{j=1}^{k} (X^T)^{k-j} \otimes X^{j-1}. \tag{1.12}$$

Since $\|A\|_F = \|\mathrm{vec}(A)\|_2$ for $A \in \mathbb{C}^{n\times n}$, we can use (1.11) to show that

$$\|L(X, \cdot)\|_F = \|K(X)\|_2 = (\|K(X)^* K(X)\|_2)^{1/2}.$$

Therefore, we can reduce the problem of estimating $\|L(X, \cdot)\|_F$ to that of estimating $\|K(X)\|_2$, which can be achieved by the power method.

**Algorithm 1.7.5.** *Given $A \in \mathbb{R}^{n\times n}$ and a nonzero vector $z_0$, the power method is used to give an approximation $\gamma$ to $\|A\|_2$.*

```
1  k = 0
2  repeat
3        w_{k+1} = A z_k
4        z_{k+1} = A^T w_{k+1}
5        k = k + 1
6  until converged
7  γ = ||z_{k+1}||_2 / ||w_{k+1}||_2
```

We now consider the power method applied to $A = K(X)^T K(X)$ for $X \in \mathbb{R}^{n\times n}$. Using (1.11) and $K(X)^T = K(X^T)$, which follows from using the property $(X \otimes Y)^T = X^T \otimes Y^T$ on (1.12), the resulting algorithm can be written in terms of $L(X, \cdot)$.

**Algorithm 1.7.6.** *Given $X \in \mathbb{R}^{n\times n}$, a nonzero matrix $Z_0 \in \mathbb{R}^{n\times n}$, a function $f$ that has a power series representation, and its Fréchet derivative $L$, the power method is used to give an approximation $\gamma$ to $\|L(X, \cdot)\|_F$.*

```
1  k = 0
2  repeat
3        W_{k+1} = L(X, Z_k)
4        Z_{k+1} = L(X^T, W_{k+1})
5        k = k + 1
6  until converged
7  γ = ||Z_{k+1}||_F / ||W_{k+1}||_F
```

An alternative to computing the potentially costly $L(X, \cdot)$ in Algorithm 1.7.6 is to use the finite difference approximation

$$L(X, E) \approx \frac{f(X + \delta E) - f(X)}{\delta}$$

for small values of $\delta$.

# Chapter 2

# Hyperbolic Transformations and the Hyperbolic QR Factorization

## 2.1 Introduction

A matrix $Q \in \mathbb{R}^{n\times n}$ is *J-orthogonal* if

$$Q^T J Q = J, \tag{2.1}$$

where the signature matrix $J = \text{diag}(\pm 1)$. We will be concerned with signature matrices of the form

$$J = \begin{bmatrix} I_p & 0 \\ 0 & -I_q \end{bmatrix}, \quad p+q = n. \tag{2.2}$$

A matrix satisfying (2.1) is sometimes referred to as *pseudo-orthogonal* [22, p. 612], [33], and when $J = \text{diag}(1, 1, 1, -1)$, physicists often refer to $Q$ as a *Lorentz* matrix [4]. Transformations that are $J$-orthogonal are often referred to as hyperbolic transformations. These hyperbolic transformations are an important tool used in many applications such as signal processing (see [1], [41]),

indefinite least squares problems (see [9], [10], [56]) and the Cholesky downdating problem (see [8]).

A matrix $Q \in \mathbb{R}^{n\times n}$ is said to be $(J_1, J_2)$-*orthogonal* if

$$Q^T J_1 Q = J_2,$$

where $J_1 = \mathrm{diag}(\pm 1)$ and $J_2 = \mathrm{diag}(\pm 1)$ have the same number of 1s and $-1$s. Transformations that are $(J_1, J_2)$-orthogonal have been used in various applications such as reducing symmetric indefinite pairs of matrices to tridiagonal-diagonal form [53] and computing the eigenvalues of pseudo-Hermitian matrices [12]. If $Q$ is complex then we say that it is $J$-*unitary* if $Q^* J Q = J$ and $(J_1, J_2)$-*unitary* if $Q^* J_1 Q = J_2$.

A link between $J$-orthogonal transformations and orthogonal transformations exists in the form of the *exchange operator*. Suppose that

$$A = \begin{matrix} n \\ \\ \end{matrix} \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \begin{matrix} p \\ q \end{matrix} = \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} \begin{matrix} n \\ \\ \end{matrix} \begin{bmatrix} B_1 \\ B_2 \end{bmatrix} \begin{matrix} p \\ q \end{matrix} = QB,$$

where $Q$ is $J$-orthogonal with $J$ given by (2.2). Then since $Q_{11}^T Q_{11} = I + Q_{21}^T Q_{21}$, $Q_{11}$ is nonsingular and

$$\begin{bmatrix} B_1 \\ A_2 \end{bmatrix} = \mathrm{exc}(Q) \begin{bmatrix} A_1 \\ B_2 \end{bmatrix},$$

where

$$\mathrm{exc}(Q) = \begin{bmatrix} Q_{11}^{-1} & -Q_{11}^{-1} Q_{12} \\ Q_{21} Q_{11}^{-1} & Q_{22} - Q_{21} Q_{11}^{-1} Q_{12} \end{bmatrix}$$

is orthogonal. If an orthogonal matrix $P$ is partitioned in the same way as $Q$ and $Q_{11}$ is nonsingular, then $\mathrm{exc}(P)$ is $J$-orthogonal. The exchange operator has the property $\mathrm{exc}(\mathrm{exc}(A)) = A$, for any $A$ and is therefore said to be *involutory*.

The exchange operator will be used extensively in later sections to analyse the errors in applying hyperbolic transformations. By applying the exchange

operator to a hyperbolic transformation we obtain an orthogonal transformation and hence error terms can be moved around without changing their norm. For more detail on the exchange operator see [31].

Although there is an extensive collection of functions for computing orthogonal transformations and factorizations in numerical libraries and software packages, there is little available for their $J$-orthogonal and $(J_1, J_2)$-orthogonal counterparts. For this reason we detail and implement a collection of hyperbolic transformations, and show how some of these may be used in the application of computing the hyperbolic QR factorization. MATLAB implementations of the methods discussed are given in Appendix A and are available at
`http://www.maths.man.ac.uk/~hargreaves/hyperbolic`.

We start in Section 2.2 with hyperbolic and unified transformations. Hyperbolic rotations analogous to Givens rotations are considered with details on how to construct them accurately. A new representation of the hyperbolic rotation is devised that avoids overflow and is shown to be stable.

We show how to apply a hyperbolic rotation directly and in mixed form, for which the latter is known to be mixed forward-backward stable. Applying a hyperbolic rotation directly is generally thought to be unstable but this has not been proved. However, we prove this is true by numerical experiments. We also consider the OD procedure of Chandrasekaran and Sayed [15] for applying hyperbolic rotations, which is known to be mixed forward-backward stable when applying one hyperbolic rotation. We prove a new error result for this method which allows us to prove stability when a sequence of hyperbolic rotations are applied in this way. A rounding error analysis for combining $r$ hyperbolic rotations is given. Unified rotations are discussed and we show how to apply these in a stable way.

In Section 2.3 we consider fast hyperbolic rotations, which require less operations than hyperbolic rotations. We take care to ensure that these are applied

in a stable way and back this up with a rounding error analysis. Hyperbolic Householder transformations are briefly discussed in Section 2.4.

The hyperbolic QR factorization of a matrix $A \in \mathbb{C}^{m\times n}$ with $m \geq n$ is

$$A = QR = Q \begin{bmatrix} R_1 \\ 0 \end{bmatrix},$$

where $Q$ is $J$-unitary and $R_1 \in \mathbb{C}^{n\times n}$ is upper triangular. In Section 2.5 we extend theorems of Bunse–Gerstner [12], to give new results for the existence of the related HR factorization, which we then use to prove new theorems for the existence of the hyperbolic QR factorization. We consider how to compute this factorization, detail the rounding error analysis, and conduct numerical experiments. Two applications for the hyperbolic QR factorization are also discussed.

## 2.2 Hyperbolic and Unified Rotations

### 2.2.1 Hyperbolic Rotations

A *hyperbolic rotation* has the form

$$H = \begin{bmatrix} \bar{c} & -\bar{s} \\ -s & c \end{bmatrix}, \quad |c|^2 - |s|^2 = 1, \tag{2.3}$$

where $|c| = \cosh(\theta)$ and $|s| = \sinh(\theta)$ for some $\theta$.

Given a vector $x \in \mathbb{C}^2$, we can choose $H$ so that $y = Hx$ is of the form $[\,\alpha \quad 0\,]^T$, provided that $|x_1| \neq |x_2|$. This is achieved by taking

$$c = \frac{x_1}{\sqrt{|x_1|^2 - |x_2|^2}}, \quad s = \frac{x_2}{\sqrt{|x_1|^2 - |x_2|^2}},$$

so that

$$y = \begin{bmatrix} \bar{c} & -\bar{s} \\ -s & c \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} \sqrt{|x_1|^2 - |x_2|^2} \\ 0 \end{bmatrix}.$$

If $x \in \mathbb{R}^2$ we would like to ensure that $c$ and $s$ are also real, which can be achieved by imposing the condition $|x_1| > |x_2|$. In the applications where

we require the use of hyperbolic rotations, it turns out that this condition is satisfied. Therefore we say that the hyperbolic rotation is not defined for $|x_1| \leq |x_2|$. We also impose this condition for $x \in \mathbb{C}^2$.

However, there are applications where we require the use of $(J_1, J_2)$-orthogonal rotations, and in Section 2.2.6 we will define such a rotation which is used to zero the second component of $x$ when $|x_1| < |x_2|$.

For a hyperbolic rotation we have that

$$\begin{bmatrix} \bar{c} & -\bar{s} \\ -s & c \end{bmatrix}^* \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} \bar{c} & -\bar{s} \\ -s & c \end{bmatrix} = \begin{bmatrix} |c|^2 - |s|^2 & 0 \\ 0 & |s|^2 - |c|^2 \end{bmatrix}$$

and hence hyperbolic rotations are $J$-unitary, where

$$J = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}.$$

One of the major issues concerning hyperbolic rotations can be demonstrated by considering the condition number of a real hyperbolic rotation. We first calculate the eigenvalues of

$$H^T H = \begin{bmatrix} c^2 + s^2 & -2cs \\ -2cs & c^2 + s^2 \end{bmatrix},$$

where $H$ is defined as in (2.3), to be

$$\lambda_1 = c^2 + s^2 + 2cs,$$
$$\lambda_2 = c^2 + s^2 - 2cs.$$

It follows that the 2-norm condition number of $H$ is

$$\kappa_2(H) = \sqrt{\frac{\max(\lambda_1, \lambda_2)}{\min(\lambda_1, \lambda_2)}} = \frac{|c| + |s|}{|c| - |s|},$$

which may be arbitrarily large, unlike for a Givens rotation whose 2-norm condition number is 1.

### 2.2.2 Representing and Applying Hyperbolic Rotations

Unlike Givens Rotations, the way in which a hyperbolic rotation is constructed and applied affects the stability of the computation. The obvious way to compute $c$ and $s$ which define the hyperbolic rotation is

$$\text{H1}: \quad c = \frac{x_1}{\sqrt{|x_1|^2 - |x_2|^2}}, \quad s = \frac{x_2}{\sqrt{|x_1|^2 - |x_2|^2}}. \tag{2.4}$$

It is often suggested that $c$ and $s$ should be scaled to avoid overflow and computed using

$$\text{H2}: \quad c = \frac{\text{sign}(x_1)}{\sqrt{1 - |t|^2}}, \quad s = tc, \quad t = \frac{x_2}{x_1}. \tag{2.5}$$

Another representation often used to compute $c$ and $s$ is

$$\text{H3}: \quad c = \frac{x_1}{\sqrt{(|x_1| + |x_2|)(|x_1| - |x_2|)}}, \quad s = \frac{x_2}{\sqrt{(|x_1| + |x_2|)(|x_1| - |x_2|)}}. \tag{2.6}$$

In Section 2.2.3 we show that this representation is preferable, for $x \in \mathbb{R}^2$, in that the error in computing $H$ using H1 or H2 is unbounded. However, using H3 does not guard against overflow.

We propose a new representation that reduces the risk of overflow, and for which the error in computing $H$, for $x \in \mathbb{R}^2$, is bounded by a small constant. We use an alternative method of computing $1 - |t|^2$ in (2.5) by first computing

$$e = \frac{|x_1| - |x_2|}{|x_1|} \in (0, 1],$$

so that $1 - |t|^2 = 2e - e^2$. This gives

$$\text{H4}: \quad c = \frac{\text{sign}(x_1)}{\sqrt{2e - e^2}}, \quad s = \frac{x_2 c}{x_1}. \tag{2.7}$$

We show the advantage of this representation over H1 and H2 in Section 2.2.3.

The following function computes $c$ and $s$ using the representation H4.

**Algorithm 2.2.1.** *Computes $c$ and $s$ that define the hyperbolic rotation $H$ so that $Hx$ has zero second component. It is assumed that $|x_1| > |x_2|$.*

1 function $[c, s] = \text{hrotate}(x)$
2 $e = (|x_1| - |x_2|)/|x_1|$
3 $c = \text{sign}(x_1)/\sqrt{2e - e^2}$
4 $s = x_2 c/x_1$

The hyperbolic rotation may be applied using various methods. It may be applied *directly* to a vector $a \in \mathbb{C}^2$ as

$$b_1 = \bar{c}a_1 - \bar{s}a_2, \tag{2.8}$$

$$b_2 = -sa_1 + ca_2. \tag{2.9}$$

However, Bojanczyk, Brent, Van Dooren and De Hoog [8] suggest the following method of applying the rotation. Solving (2.8) for $a_1$ gives

$$a_1 = \frac{b_1 + \bar{s}a_2}{\bar{c}} \tag{2.10}$$

and hence (2.9) can be written as

$$\begin{aligned} b_2 &= -\frac{s}{\bar{c}}b_1 + \left(-\frac{|s|^2}{\bar{c}} + c\right) a_2, \\ &= -\frac{s}{\bar{c}}b_1 + \frac{a_2}{\bar{c}}. \end{aligned} \tag{2.11}$$

Applying the hyperbolic rotation using (2.8) and (2.11) is referred to as applying the rotation in *mixed form* and is generally thought to be more stable than applying the rotation directly. In the case of applying a Givens rotation $G \in \mathbb{C}^{2\times 2}$ to $a \in \mathbb{C}^2$ to obtain $b = Ga$, Gill, Golub, Murray and Saunders [21] have shown how to use $b_1$ in the computation of $b_2$ in order to reduce the number of multiplications required in applying $G$.

The equations (2.10) and (2.11) can be combined so that

$$\begin{bmatrix} a_1 \\ b_2 \end{bmatrix} = G \begin{bmatrix} b_1 \\ a_2 \end{bmatrix},$$

where

$$G = \begin{bmatrix} 1/\bar{c} & \bar{s}/\bar{c} \\ -s/\bar{c} & 1/\bar{c} \end{bmatrix}$$

is unitary. The condition $|x_1| > |x_2|$ ensures that $c \neq 0$ and hence the exchange operator can be used on any hyperbolic rotation, and also $\mathrm{exc}(\mathrm{exc}(H)) = H$.

The following function applies the hyperbolic rotation to a $2 \times n$ matrix in mixed form.

**Algorithm 2.2.2.** *Applies the hyperbolic rotation, $H$, defined by $c$ and $s$ to a $2 \times n$ matrix $A$ in mixed form so that $B = HA$.*

```
1  function B = happly(c, s, A)
2  B(1,:) = c̄A(1,:) − s̄A(2,:)
3  B(2,:) = (−sB(1,:) + A(2,:))/c̄
```

An alternative method of applying a real hyperbolic rotation is the OD (Orthogonal-Diagonal) procedure of Chandrasekaran and Sayed [15], which expresses the hyperbolic rotation that zeros the second component of $x \in \mathbb{R}^2$ as

$$\text{H5}: \quad H = QDQ^T = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} d/2 & 0 \\ 0 & 1/(2d) \end{bmatrix} \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}, \quad d = \sqrt{\frac{x_1 + x_2}{x_1 - x_2}}. \tag{2.12}$$

We note that removing a factor of $1/2$ from $D$ and multiplying $Q$ by $1/\sqrt{2}$ gives the singular value decomposition of

$$H = \begin{bmatrix} c & -s \\ -s & c \end{bmatrix},$$

and that the elements $c$ and $s$ are not computed explicitly, but $H$ is applied to $A \in \mathbb{R}^{2\times n}$ as $B = Q(D(Q^T A))$. This can be efficiently implemented as described by the following function.

**Algorithm 2.2.3.** *Applies a hyperbolic rotation that zeroes the second component of $x \in \mathbb{R}^2$ using the OD procedure, to a $2 \times n$ matrix $A$ to obtain $B$.*

```
1  function B = happly_od(x, A)
2  d = sqrt((x1+x2)/(x1−x2))
```

3 $\widetilde{B}(1,:) = A(1,:) - A(2,:)$, $\widetilde{B}(2,:) = A(1,:) + A(2,:)$
4 $\widetilde{B}(1,:) = (d/2)\widetilde{B}(1,:)$, $\widetilde{B}(1,:) = (1/(2d))\widetilde{B}(2,:)$
5 $B(1,:) = \widetilde{B}(1,:) + \widetilde{B}(2,:)$, $B(2,:) = -\widetilde{B}(1,:) + \widetilde{B}(2,:)$

Applying a hyperbolic rotation to a $2 \times n$ matrix using the OD procedure requires $6n$ operations, which is same as the cost of applying a hyperbolic rotation directly or in mixed form.

The OD procedure has good numerical properties, which result from the fact that the hyperbolic rotation is applied as a sequence of orthogonal and diagonal matrices, and is shown by Chandrasekaran and Sayed [15] to be mixed forward-backward stable when applying one hyperbolic rotation in this way. This result is described in Section 2.2.3 together with a new error result which will allow us to perform a rounding error analysis for a sequence of hyperbolic rotations applied using the OD procedure.

MATLAB implementations of the methods described in this section for constructing and applying hyperbolic rotations are given in Appendix A.1.

### 2.2.3 Error Analysis for Hyperbolic Rotations

Here we perform a rounding error analysis for forming a hyperbolic rotation and applying it to a vector. We first study the errors in computing $c$ and $s$ using the representations described in Section 2.2.2. We restrict the analysis to the real case but provide numerical experiments to show the problems with computing $c$ and $s$ using the representations described in Section 2.2.2 in the complex case.

**Lemma 2.2.4.** *Let $c$ and $s$ defining a hyperbolic rotation be computed using H1–H4. Then the errors $\Delta c = fl(c) - c$ and $\Delta s = fl(s) - s$ are bounded by*

$$\text{H1}: \quad |\Delta c| \le \left( \frac{1+\gamma_2}{\sqrt{1 - \gamma_2 \frac{x_1^2+x_2^2}{x_1^2-x_2^2}}} - 1 \right) |c|, \quad |\Delta s| \le \left( \frac{1+\gamma_2}{\sqrt{1 - \gamma_2 \frac{x_1^2+x_2^2}{x_1^2-x_2^2}}} - 1 \right) |s|,$$

$$\text{H2}: \quad |\Delta c| \le \left( \frac{1+\gamma_2}{\sqrt{1-\gamma_4 \frac{x_1^2+x_2^2}{x_1^2-x_2^2}}} - 1 \right) |c|, \quad |\Delta s| \le \left( \frac{1+\gamma_4}{\sqrt{1-\gamma_4 \frac{x_1^2+x_2^2}{x_1^2-x_2^2}}} - 1 \right) |s|,$$

$$\text{H3}: \quad |\Delta c| \le \gamma_5 |c|, \quad |\Delta s| \le \gamma_5 |s|,$$

$$\text{H4}: \quad |\Delta c| \le \gamma_{18} |c|, \quad |\Delta s| \le \gamma_{20} |s|.$$

*Proof.* For a proof of the results for H1–H3 see [42].

Here we consider H4. Recall that $e = (|x_1| - |x_2|)/|x_1|$, from which $\widehat{e} = e(1+\theta_2)$ is immediate. Using the fact that multiplying a floating point number by 2 on a binary machine is exact, we then have

$$\begin{aligned} fl(c) &= \frac{1+\delta_4}{\sqrt{(2\widehat{e} - \widehat{e}^2(1+\delta_1))(1+\delta_2)(1+\delta_3)}} \\ &= \frac{1}{\sqrt{2e-e^2}} \left( \frac{(1+\theta_2)\sqrt{2e-e^2}}{\sqrt{2e(1+\theta_1) - e^2(1+\theta_2')}} \right) \\ &= \frac{1}{\sqrt{2e-e^2}} (1+\theta_2) \left( 1 + \frac{2e\theta_1 - e^2\theta_2'}{2e-e^2} \right)^{-1/2} \\ &= \frac{1}{\sqrt{2e-e^2}} (1+\theta_2)(1+\epsilon_1)^{-1/2}, \end{aligned}$$

where

$$|\epsilon_1| \le \gamma_3 \frac{2e+e^2}{2e-e^2} \le \gamma_3 \frac{1}{e}(2e+e^2) \le 3\gamma_3 \le \gamma_9.$$

Hence

$$fl(c) = c(1+\eta_1),$$

where $\eta_1 = (1+\theta_2)(1+\epsilon_1)^{-1/2} - 1$ and

$$\begin{aligned} |\eta_1| &\le \frac{1+\gamma_2}{\sqrt{1-\gamma_9}} - 1 \\ &\le \frac{1+\gamma_9}{1-\gamma_9} - 1 \\ &\le 1 + \gamma_{18} - 1 \quad \text{(a consequence of Lemma 1.4.2)} \\ &= \gamma_{18}. \end{aligned}$$

We also obtain

$$fl(s) = s(1+\eta_2),$$

where $|\eta_2| \leq \gamma_{20}$. $\square$

Lemma 2.2.4 suggests that H3 and H4 can be computed more accurately than H1 and H2, although if $|x_1|$ is not close to $|x_2|$ all four representations will be accurate. This is supported by numerical experiments. Using MATLAB we computed $\|H - fl(H)\|_2/\|H\|_2$ for H1–H4, for a series of vectors $x \in \mathbb{R}^2$ with $x_2$ approaching $x_1$ where $H$ is calculated to 100 significant digits. The results are given in Table 2.1, and show the benefit of the H3 and H4 representations. We therefore do not further consider or make use of the H1 and H2 representations.

Table 2.1: The errors $\|H - fl(H)\|_2/\|H\|_2$ in forming $H$ by H1–H4 using the vector $x = [5000, 5000 - \alpha]^T$.

| $\alpha$ | H1 | H2 | H3 | H4 |
|---|---|---|---|---|
| 1e-00 | 4.6280e-17 | 2.8732e-14 | 4.6280e-17 | 4.6280e-17 |
| 1e-02 | 7.5126e-13 | 6.4709e-12 | 1.8665e-17 | 9.5022e-17 |
| 1e-04 | 2.6335e-10 | 1.8257e-09 | 5.1384e-17 | 5.1384e-17 |
| 1e-06 | 8.7261e-08 | 1.2785e-07 | 1.3048e-16 | 5.7722e-17 |
| 1e-08 | 4.8750e-06 | 5.7733e-06 | 4.5305e-17 | 2.1602e-17 |
| 1e-10 | 8.2806e-04 | 6.2203e-04 | 3.7077e-17 | 3.7077e-17 |

We now consider computing $c$ and $s$ using representations H3 and H4 for $x \in \mathbb{C}^2$. For both representations we must compute

$$\begin{aligned} r &= |x_1| - |x_2| \\ &= \sqrt{a^2 + b^2} - \sqrt{e^2 + f^2}, \end{aligned} \tag{2.13}$$

where $x_1 \equiv a + ib$ and $x_2 \equiv e + if$, and using (2.13) may cause cancellation similar to that observed in the computation of H1 and H2 in the real case. This is highlighted by numerical experiments. We computed $\|H - fl(H)\|_2/\|H\|_2$ using H3 and H4 for a series of vectors $x \in \mathbb{C}^2$, with $x_2$ approaching $x_1$, where $H$ is calculated to 100 significant digits. The results in Table 2.2 show the instability of using H3 or H4 for $x \in \mathbb{C}^2$.

An alternative method of computing $c$ and $s$ for $x \in \mathbb{C}^2$ is to compute the denominator $d = \sqrt{|x_1|^2 - |x_2|^2}$ using

$$d = \sqrt{(a+e)(a-e) + (b+f)(b-f)}. \tag{2.14}$$

This removes the risk of cancellation when $\text{sign}(a^2 - e^2) = \text{sign}(b^2 - f^2)$, but may still suffer from cancellation otherwise. Similarly, using

$$d = \sqrt{(a+f)(a-f) + (b+e)(b-e)} \tag{2.15}$$

will not suffer from cancellation if $\text{sign}(a^2 - f^2) = \text{sign}(b^2 - e^2)$. We can choose to use (2.14) or (2.15), depending on the circumstances, so that the risk of cancellation is significantly reduced. It is possible to ensure that cancellation may only occur if $|a|$, $|b|$, $|e|$ and $|f|$ are close together, $\text{sign}(a^2 - e^2) = -\text{sign}(b^2 - f^2)$ and $\text{sign}(a^2 - f^2) = -\text{sign}(a^2 - e^2)$. However, since we are unable to find any stable methods to compute $c$ and $s$ for *all* $|x_1| > |x_2|$, we restrict our attention to the real case.

Table 2.2: The errors $\|H - fl(H)\|_2 / \|H\|_2$ in forming $H$ by H3 and H4 for the vector $x = [3000 + 2000i, (3000 - \alpha) + (2000 - \alpha)i]^T$.

| $\alpha$ | H3 | H4 |
|---|---|---|
| 1e-00 | 4.3008e-14 | 4.3036e-14 |
| 1e-02 | 6.9506e-12 | 6.9505e-12 |
| 1e-04 | 9.0943e-11 | 9.0943e-11 |
| 1e-06 | 8.1762e-08 | 8.1762e-08 |
| 1e-08 | 5.8488e-06 | 5.8488e-06 |
| 1e-10 | 1.3949e-04 | 1.3949e-04 |

Next we consider the errors in applying $H \in \mathbb{R}^{2\times 2}$, represented by H3 and H4, directly to a vector $a \in \mathbb{R}^2$.

**Lemma 2.2.5.** *Let a hyperbolic rotation $H$ be formed by H3 or H4 and applied directly to $a = [\, a_1 \quad a_2 \,]^T$, giving $b = Ha$. The computed $\widehat{b}$ satisfies*

$$\text{H3}: \quad \widehat{b} = H(a + \Delta a), \quad |\Delta a| \le \gamma_7 |H|^2 |a|,$$

$$\text{H4}: \quad \widehat{b} = H(a + \Delta a), \quad |\Delta a| \leq \gamma_{22}|H|^2|a|.$$

*Proof.* We prove the result for H4. The proof for H3 is similar.

From Lemma 2.2.4 we have that

$$fl(c) = c(1 + \eta_1), \quad |\eta_1| \leq \gamma_{18},$$
$$fl(s) = s(1 + \eta_2), \quad |\eta_2| \leq \gamma_{20}.$$

Hence

$$\widehat{b}_1 = (c(1 + \eta_1)a_1(1 + \delta_1) - s(1 + \eta_2)a_2(1 + \delta_2))(1 + \delta_3) \tag{2.16}$$
$$= ca_1(1 + \epsilon_1) - sa_2(1 + \epsilon_2), \tag{2.17}$$

where

$$|\epsilon_1| = |\eta_1 + \theta_2 + \eta_1\theta_2| \leq \gamma_{18} + \gamma_2 + \gamma_{18}\gamma_2 \leq \gamma_{20},$$

and similarly

$$|\epsilon_2| \leq \gamma_{22}.$$

We can also find that

$$\widehat{b}_2 = -sa_1(1 + \epsilon_3) + ca_2(1 + \epsilon_4), \tag{2.18}$$

where

$$|\epsilon_3| \leq \gamma_{22}, \quad |\epsilon_4| \leq \gamma_{20}.$$

Combining (2.17) and (2.18) we obtain

$$\widehat{b} = \begin{bmatrix} \widehat{b}_1 \\ \widehat{b}_2 \end{bmatrix} = (H + \Delta H)a, \quad |\Delta H| \leq \gamma_{22}|H|,$$

and therefore

$$\widehat{b} = H(a + \Delta a), \quad |\Delta a| \leq \gamma_{22}|H||H^{-1}||a| = \gamma_{22}|H|^2|a|. \qquad \square$$

For hyperbolic rotations applied directly, we have been unable to find error bounds that do not depend on $H$. Since the components of $H$ may be arbitrarily large it seems that applying hyperbolic rotations directly may be unstable. Fortunately, hyperbolic rotations can be applied in a stable way by applying them in mixed form.

**Lemma 2.2.6.** *Let a hyperbolic rotation, $H$, constructed using H3 or H4 be applied in mixed form. Let $a = [a_1 \ a_2]^T \in \mathbb{R}^2$ and $b = [b_1 \ b_2]^T = Ha$. Then the computed $\widehat{b}$ satisfies*

$$\widehat{b} + \Delta b = H(a + \Delta a),$$

*where* $\Delta b = \begin{bmatrix} \Delta b_1 \\ 0 \end{bmatrix}$, $\Delta a = \begin{bmatrix} 0 \\ \Delta a_2 \end{bmatrix}$ *and*

$$\text{H3}: \quad \max(|\Delta b_1|, |\Delta a_2|) \leq \gamma_{20} \max(|\widehat{b}_1|, |a_2|),$$

$$\text{H4}: \quad \max(|\Delta b_1|, |\Delta a_2|) \leq \gamma_{82} \max(|\widehat{b}_1|, |a_2|).$$

*Proof.* We will prove the result for H4. The result for H3 is proved in a similar way.

From Lemma 2.2.4 we have that

$$fl(c) = c(1 + \eta_1), \quad |\eta_1| \leq \gamma_{18},$$

$$fl(s) = s(1 + \eta_2), \quad |\eta_2| \leq \gamma_{20}.$$

The first component of $b$ is computed in the same way as when the hyperbolic rotation is applied directly and therefore satisfies (2.16), which can be rewritten as

$$a_1 = \frac{\widehat{b}_1}{c}(1 + \epsilon_1) + \frac{s}{c}a_2(1 + \epsilon_2), \tag{2.19}$$

where $|\epsilon_1| \leq \gamma_{40}$ and $|\epsilon_2| \leq \gamma_{40}$. The computed second component of $b$ satisfies

$$\widehat{b}_2 = -\frac{s}{c}\widehat{b}_1(1 + \epsilon_3) + \frac{a_2}{c}(1 + \epsilon_4), \tag{2.20}$$

where $|\epsilon_3| \leq \gamma_{41}$ and $|\epsilon_4| \leq \gamma_{38}$.

Combining (2.19) and (2.20), and using $|\epsilon_1|, |\epsilon_2|, |\epsilon_3|, |\epsilon_4| \leq \gamma_{41}$ we obtain

$$\begin{bmatrix} a_1 \\ \widehat{b}_2 \end{bmatrix} = (G + \Delta G) \begin{bmatrix} \widehat{b}_1 \\ a_2 \end{bmatrix}, \quad |\Delta G| \leq \gamma_{41}|G|,$$

where

$$G = \begin{bmatrix} \widetilde{c} & \widetilde{s} \\ -\widetilde{s} & \widetilde{c} \end{bmatrix}, \quad \widetilde{c} = 1/c, \ \widetilde{s} = s/c,$$

is orthogonal. This can be rewritten as

$$\begin{bmatrix} a_1 \\ \widehat{b}_2 \end{bmatrix} = G \begin{bmatrix} \widehat{b}_1 + \Delta b_1 \\ a_2 + \Delta a_2 \end{bmatrix}, \tag{2.21}$$

with

$$\begin{bmatrix} \Delta b_1 \\ \Delta a_2 \end{bmatrix} = G^T \Delta G \begin{bmatrix} \widehat{b}_1 \\ a_2 \end{bmatrix}$$

and

$$\begin{aligned} \max(|\Delta b_1|, |\Delta a_2|) &\leq \gamma_{41}(1 + 2|\widetilde{c}||\widetilde{s}|) \max(|\widehat{b}_1|, |a_2|) \\ &\leq \gamma_{82} \max(|\widehat{b}_1|, |a_2|), \end{aligned}$$

where $\widetilde{c} = 1/c$ and $\widetilde{s} = s/c$. Using the exchange operator we can rewrite (2.21) as

$$\begin{bmatrix} \widehat{b}_1 + \Delta b_1 \\ \widehat{b}_2 \end{bmatrix} = H \begin{bmatrix} a_1 \\ a_2 + \Delta a_2 \end{bmatrix}. \qquad \square$$

This result is a mixed forward-backward error result, and the errors do not depend on the size of $H$. It shows that applying a hyperbolic rotation in mixed form is mixed forward-backward stable. The OD procedure described by H5 for applying hyperbolic rotations has also been shown, by Chandrasekaran and Sayed [15], to be mixed forward-backward stable. Their result, also proved in [42], is summarised in the following lemma.

**Lemma 2.2.7.** *Let a hyperbolic rotation, $H = QDQ^T$, be expressed in the form* (2.12). *Let $a = [a_1 \ a_2]^T \in \mathbb{R}^2$ and $b = [b_1 \ b_2]^T = Ha$. Then if $b$ is formed using $b = Q(D(Q^T a))$, the computed $\widehat{b}$ satisfies*

$$\widehat{b} + \Delta b = H(a + \Delta a),$$

*where* $|\Delta b| \le \gamma_1 |\widehat{b}|$ *and* $|\Delta a| \le 2\gamma_6 |Q||a|$. *Normwise bounds are* $\|\Delta b\|_2 \le \gamma_1 \|\widehat{b}\|_2$ *and* $\|\Delta a\|_2 \le 4\gamma_6 \|a\|_2$.

*Proof.* Let $b' = Q^T a$, $b'' = Db'$ and $b = Qb''$. Then the computed $b'$ satisfies

$$\begin{aligned}
\widehat{b'} &= fl(Q^T a) \\
&= \begin{bmatrix} 1+\delta_1 & 0 \\ 0 & 1+\delta_2 \end{bmatrix} \begin{bmatrix} a_1 - a_2 \\ a_1 + a_2 \end{bmatrix}, \quad |\delta_1|, |\delta_2| \le u, \\
&= (I + \Delta_1) Q^T a, \quad |\Delta_1| \le \gamma_1 I.
\end{aligned}$$

The computed $d$ satisfies

$$\begin{aligned}
\widehat{d} &= fl\left(\sqrt{\frac{x_1 + x_2}{x_1 - x_2}}\right) \\
&= \sqrt{\frac{(x_1 + x_2)(1+\delta_3)}{(x_1 - x_2)(1+\delta_4)}(1+\delta_5)(1+\delta_6)} \\
&= d(1+\theta_4),
\end{aligned}$$

and since division by 2 can be considered to be done exactly, $\widehat{d/2} = d(1+\theta_4)/2$ and $\widehat{1/(2d)} = (1+\theta_4)/(2d)$. The computed $b''$ therefore satisfies

$$\begin{aligned}
\widehat{b''} &= fl(D\widehat{b'}) \\
&= D(I + \Delta_2)(I + \Delta_1) Q^T a, \quad |\Delta_2| \le \gamma_5 I, \\
&= D(I + \Delta_3) Q^T a,
\end{aligned}$$

where $|\Delta_3| \le \gamma_6 I$. Applying $Q$ to $\widehat{b''}$ gives

$$\begin{aligned}
\widehat{b} &= fl(Q\widehat{b''}) \\
&= (I + \Delta_4) QD(I + \Delta_3) Q^T a,
\end{aligned}$$

where $|\Delta_4| \le \gamma_1 I$. Rearranging and using $Q^T Q = I$ gives

$$\begin{aligned}
(I + \Delta_5)\widehat{b} &= QD(I + \Delta_3) Q^T a, \quad |\Delta_5| \le \gamma_1 I, \\
&= QDQ^T (I + Q\Delta_3 Q^T) a \\
&= QDQ^T (I + \Delta_6) a,
\end{aligned}$$

where

$$|\Delta_6| = |Q\Delta_3 Q^T| \leq \gamma_6 |Q|^2 = 2\gamma_6 |Q|.$$

Hence we have the result

$$\widehat{b} + \Delta b = H(a + \Delta a), \quad |\Delta b| \leq \gamma_1 |\widehat{b}|, \quad |\Delta a| \leq 2\gamma_6 |Q||a|,$$

and the normwise bounds follow using $\||Q|\|_2 \leq 2$. □

We now seek a new mixed forward-backward error result that has no perturbation for the $b_2$ and $a_1$ components. This will later allow us to conduct a rounding error analysis when more than one hyperbolic rotation is applied using the OD procedure. The following lemma shows how we can remove the perturbations from the $b_2$ and $a_1$ components for a more general mixed forward-backward error result.

**Lemma 2.2.8.** *Let $a = [a_1\ a_2]^T \in \mathbb{R}^2$, $b = [b_1\ b_2]^T = Ha$ and $H$ be a hyperbolic rotation that satisfies*

$$\begin{bmatrix} \widehat{b}_1 + \Delta b_1 \\ \widehat{b}_2 + \Delta b_2 \end{bmatrix} = H \begin{bmatrix} a_1 + \Delta a_1 \\ a_2 + \Delta a_2 \end{bmatrix}, \tag{2.22}$$

*where*

$$|\Delta b_1| \leq \epsilon_1 |\widehat{b}_1|, \quad |\Delta b_2| \leq \epsilon_2 |\widehat{b}_2|, \quad |\Delta a_1| \leq \epsilon_3 |a_1|, \quad |\Delta a_2| \leq \epsilon_4 |a_2|.$$

*Then*

$$\begin{bmatrix} \widehat{b}_1 + \Delta_1 \\ \widehat{b}_2 \end{bmatrix} = H \begin{bmatrix} a_1 \\ a_2 + \Delta_2 \end{bmatrix},$$

*where*

$$|\Delta_1| \leq (\epsilon_1 + 2\epsilon_2 + 2\epsilon_3) \max(|\widehat{b}_1|, |a_2|),$$
$$|\Delta_2| \leq (\epsilon_4 + 2\epsilon_2 + 2\epsilon_3) \max(|\widehat{b}_1|, |a_2|),$$

*Proof.* Applying the exchange operator to (2.22) gives

$$\begin{bmatrix} a_1 + \Delta a_1 \\ \widehat{b}_2 + \Delta b_2 \end{bmatrix} = G \begin{bmatrix} \widehat{b}_1 + \Delta b_1 \\ a_2 + \Delta a_2 \end{bmatrix}, \tag{2.23}$$

where

$$G = \begin{bmatrix} 1/c & s/c \\ -s/c & 1/c \end{bmatrix}.$$

Since $|s/c|, |1/c| < 1$, we have

$$\begin{aligned} |\widehat{b}_2| &\leq |s/c||\widehat{b}_1 + \Delta b_1| + |1/c||a_2 + \Delta a_2| + |\Delta b_2| \\ &\leq |\widehat{b}_1| + |\Delta b_1| + |a_2| + |\Delta a_2| + |\Delta b_2| \\ &\leq 2\max(|\widehat{b}_1|, |a_2|) + O(u). \end{aligned}$$

Similarly

$$|a_1| \leq 2\max(|\widehat{b}_1|, |a_2|) + O(u).$$

Ignoring second order terms of $u$ we now have the bounds

$$\begin{aligned} |\Delta b_2| &\leq \epsilon_2 |\widehat{b}_2| \leq 2\epsilon_2 \max(|\widehat{b}_1|, |a_2|), \\ |\Delta a_1| &\leq \epsilon_3 |a_1| \leq 2\epsilon_3 \max(|\widehat{b}_1|, |a_2|). \end{aligned}$$

The perturbations on the left hand side of (2.23) can be rearranged so that

$$\begin{bmatrix} a_1 \\ \widehat{b}_2 \end{bmatrix} = G \begin{bmatrix} \widehat{b}_1 + \Delta b_1 \\ a_2 + \Delta a_2 \end{bmatrix} - GG^T \begin{bmatrix} \Delta a_1 \\ \Delta b_2 \end{bmatrix}.$$

Since

$$G^T \begin{bmatrix} \Delta a_1 \\ \Delta b_2 \end{bmatrix} = \begin{bmatrix} (1/c)\Delta a_1 + (s/c)\Delta b_2 \\ -(s/c)\Delta a_1 + (1/c)\Delta b_2 \end{bmatrix},$$

and $|s/c|, |1/c| < 1$, we can write

$$\begin{bmatrix} a_1 \\ \widehat{b}_2 \end{bmatrix} = G \begin{bmatrix} \widehat{b}_1 + \Delta_1 \\ a_2 + \Delta_2 \end{bmatrix},$$

where

$$\begin{aligned} |\Delta_1| &\leq |\Delta b_1| + |\Delta a_1| + |\Delta b_2| \leq (\epsilon_1 + 2\epsilon_3 + 2\epsilon_2)\max(|\widehat{b}_1|, |a_2|), \\ |\Delta_2| &\leq |\Delta a_2| + |\Delta a_1| + |\Delta b_2| \leq (\epsilon_4 + 2\epsilon_3 + 2\epsilon_2)\max(|\widehat{b}_1|, |a_2|). \end{aligned}$$

Applying the exchange operator we get

$$\begin{bmatrix} \widehat{b}_1 + \Delta_1 \\ \widehat{b}_2 \end{bmatrix} = H \begin{bmatrix} a_1 \\ a_2 + \Delta_2 \end{bmatrix},$$

with the required bounds for $|\Delta_1|$ and $|\Delta_2|$. □

**Corollary 2.2.9.** *Let a hyperbolic rotation, $H = QDQ^T$, be expressed in the form* (2.12)*. Let $a = [a_1 \ a_2]^T \in \mathbb{R}^2$ and $b = [b_1 \ b_2]^T = Ha$. Then if $b$ is formed using $b = Q(D(Q^T a))$, the computed $\widehat{b}$ satisfies*

$$\begin{bmatrix} \widehat{b}_1 + \Delta_1 \\ \widehat{b}_2 \end{bmatrix} = H \begin{bmatrix} a_1 \\ a_2 + \Delta_2 \end{bmatrix},$$

*where* $\max(|\Delta_1|, |\Delta_2|) \leq 14\gamma_6 \max(|\widehat{b}_1|, |a_2|)$.

*Proof.* Since $|\Delta b| \leq \gamma_1 |\widehat{b}|$ and $|\Delta a| \leq 2\gamma_6 |Q||a|$ in Lemma 2.2.7, we have

$$\begin{bmatrix} \widehat{b}_1 + \Delta b_1 \\ \widehat{b}_2 + \Delta b_2 \end{bmatrix} = H \begin{bmatrix} a_1 + \Delta a_1 \\ a_2 + \Delta a_2 \end{bmatrix},$$

where

$$|\Delta b_1| \leq \gamma_1 |\widehat{b}_1|,$$

$$|\Delta b_2| \leq \gamma_1 |\widehat{b}_2|,$$

$$|\Delta a_1|, |\Delta a_2| \leq 4\gamma_6 \max(|a_1|, |a_2|).$$

By Lemma 2.2.8 we have

$$\begin{bmatrix} \widehat{b}_1 + \Delta_1 \\ \widehat{b}_2 \end{bmatrix} = H \begin{bmatrix} a_1 \\ a_2 + \Delta_2 \end{bmatrix},$$

where

$$|\Delta_1| \leq 11\gamma_6 \max(|\widehat{b}_1|, |a_2|), \quad |\Delta_2| \leq 14\gamma_6 \max(|\widehat{b}_1|, |a_2|),$$

and hence we have the desired result. □

Corollary 2.2.9 shows that the errors do not depend on the size of $H$, and we note that the error result is similar to that for the mixed method of applying a hyperbolic rotation as described by Lemma 2.2.6.

### 2.2.4 Numerical Experiments

Let us consider applying the hyperbolic rotation $H \in \mathbb{R}^{2\times 2}$, such that $Hx$ has zero second component, to $a \in \mathbb{R}^2$ so that $b = Ha$. If a method of applying a

hyperbolic rotation is mixed forward-backward stable then there should exist a hyperbolic rotation $\widetilde{H}$ such that

$$\begin{bmatrix} \widehat{b}_1 + \Delta_1 \\ \widehat{b}_2 \end{bmatrix} = \widetilde{H} \begin{bmatrix} a_1 \\ a_2 + \Delta_2 \end{bmatrix}, \tag{2.24}$$

with $\|[\Delta_1 \ \Delta_2]^T\|_2$ close to

$$\delta = u \left\| \begin{bmatrix} \widehat{b}_1 \\ a_2 \end{bmatrix} \right\|_2, \tag{2.25}$$

where $u$ is the unit roundoff. We have shown in Lemma 2.2.6 and Corollary 2.2.9 that such a matrix $\widetilde{H}$ exists when applying a hyperbolic rotation in mixed form or by the OD procedure. In Section 2.2.3 we suggested that applying a hyperbolic rotation directly may be unstable and we now confirm this by numerical experiments.

To check the stability of applying a hyperbolic rotation it is sufficient to check whether we can find a mixed forward-backward error result of the form (2.24), since the proof of Lemma 2.2.8 shows how we can convert an error result of the form (2.22) to the form (2.24). Using the exchange operator we can rewrite (2.24) as

$$G \begin{bmatrix} \widehat{b}_1 + \Delta_1 \\ a_2 + \Delta_2 \end{bmatrix} = \begin{bmatrix} a_1 \\ \widehat{b}_2 \end{bmatrix}, \tag{2.26}$$

where

$$G = \operatorname{exc}(\widetilde{H}) = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}, \quad c^2 + s^2 = 1,$$

is orthogonal. Rearranging we find that

$$\begin{aligned} \begin{bmatrix} \Delta_1 \\ \Delta_2 \end{bmatrix} &= G^T \begin{bmatrix} a_1 \\ \widehat{b}_2 \end{bmatrix} - \begin{bmatrix} \widehat{b}_1 \\ a_2 \end{bmatrix} \\ &= \begin{bmatrix} a_1 & -\widehat{b}_2 \\ \widehat{b}_2 & a_1 \end{bmatrix} \begin{bmatrix} c \\ s \end{bmatrix} - \begin{bmatrix} \widehat{b}_1 \\ a_2 \end{bmatrix}. \end{aligned}$$

Hence, we can find the values of $c$ and $s$ that give the minimum value of $\|[\Delta_1 \ \Delta_2]^T\|_2$ in (2.26) by solving the constrained least squares problem

$$\min_y \left( \left\| \begin{bmatrix} a_1 & -\widehat{b}_2 \\ \widehat{b}_2 & a_1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} - \begin{bmatrix} \widehat{b}_1 \\ a_2 \end{bmatrix} \right\|_2 \right) \quad \text{subject to } \|y\|_2 = 1. \tag{2.27}$$

An algorithm for solving such problems is detailed in [22, Sec. 12.1].

For various $x \in \mathbb{R}^2$ and $a \in \mathbb{R}^2$ we computed $b = Ha$, where $H$ is the hyperbolic rotation such that $Hx$ has zero second component, by applying $H$ directly, in mixed form and by the OD procedure. We solved (2.27) for each method in 100-digit arithmetic using MATLAB's Symbolic Math Toolbox.

We recall that we would expect the minimum in (2.27) to be close to (2.25) if a method of applying $H$ is stable. Therefore the first four results in Table 2.3 prove that applying a hyperbolic rotation directly is unstable. For this reason we no longer consider applying a hyperbolic rotation using this method. As expected, a hyperbolic rotation applied in mixed form or by the OD procedure behaves in a stable way.

Table 2.3: The value (2.27), computed for a hyperbolic rotation applied directly, in mixed form, and by the OD procedure. The hyperbolic rotation, $H$, such that $Hx$ has zero second component, where $x = [1, 1-\alpha]$, is applied to $a = [5, 5-\beta]$. The value $\delta$ is given by (2.25).

| $\alpha$ | $\beta$ | $\delta$ | Direct | Mixed | OD |
|---|---|---|---|---|---|
| 1.0e-08 | 1.0e-02 | 7.9e-15 | 3.8e-12 | 8.2e-15 | 8.1e-15 |
| 1.0e-12 | 1.0e-02 | 7.9e-13 | 1.6e-10 | 5.9e-13 | 5.9e-13 |
| 1.0e-12 | 1.0e-04 | 7.9e-15 | 6.1e-10 | 8.1e-15 | 6.1e-15 |
| 1.0e-12 | 1.0e-08 | 5.6e-16 | 7.7e-13 | 1.4e-16 | 2.8e-22 |
| 1.0e-02 | 1.0e-08 | 5.6e-16 | 3.9e-16 | 3.9e-16 | 2.4e-18 |
| 1.0e-04 | 1.0e-12 | 5.6e-16 | 2.8e-17 | 2.2e-16 | 1.7e-20 |

### 2.2.5 Combining Hyperbolic Rotations

Here we analyse a product of hyperbolic rotations that are nonoverlapping in components $1\colon p$. Following Bojanczyk, Higham and Patel [9], we say that two hyperbolic transformations are nonoverlapping in components $1\colon p$ (or nonoverlapping for short) if for $i = 1\colon p$ at least one of the transformations agrees with the identity in row $i$ and column $i$. This will allow us to conduct a rounding

error analysis of algorithms which apply nonoverlapping hyperbolic rotations.

Consider first a product of two hyperbolic transformations, which are $J$-orthogonal, where $J = \mathrm{diag}(I_p, -I_q)$ and $p + q = m$. We will require the following lemma to write our error results in terms of the original hyperbolic transformations.

**Lemma 2.2.10.** *Consider two hyperbolic transformations, $H_1 \in \mathbb{R}^{m\times m}$ agreeing with the identity matrix in rows and columns $1{:}\,t$, and $H_2 \in \mathbb{R}^{m\times m}$ agreeing with the identity matrix in rows and columns $t+1{:}\,p$, where $1 \le t \le p$. If $\mathrm{exc}(H_1) = G_1$ and $\mathrm{exc}(H_2) = G_2$ then*

$$\mathrm{exc}(H_2H_1) = G_2G_1$$

*and*

$$\mathrm{exc}(G_2G_1) = H_2H_1.$$

*Proof.* Applying the exchange operator to the hyperbolic transformations

$$H_1 \equiv \begin{array}{c} \\ t \\ p-t \\ m-p \end{array}\!\!\begin{array}{c} \begin{array}{ccc} t & p-t & m-p \end{array} \\ \begin{bmatrix} I & 0 & 0 \\ 0 & A_{11} & A_{12} \\ 0 & A_{21} & A_{22} \end{bmatrix} \end{array}, \quad H_2 \equiv \begin{array}{c} \\ t \\ p-t \\ m-p \end{array}\!\!\begin{array}{c} \begin{array}{ccc} t & p-t & m-p \end{array} \\ \begin{bmatrix} B_{11} & 0 & B_{12} \\ 0 & I & 0 \\ B_{21} & 0 & B_{22} \end{bmatrix} \end{array},$$

gives

$$G_1 = \mathrm{exc}(H_1) = \begin{bmatrix} I & 0 & 0 \\ 0 & A_{11}^{-1} & -A_{11}^{-1}A_{12} \\ 0 & A_{21}A_{11}^{-1} & A_{22} - A_{21}A_{11}^{-1}A_{12} \end{bmatrix}, \tag{2.28}$$

and

$$G_2 = \mathrm{exc}(H_2) = \begin{bmatrix} B_{11}^{-1} & 0 & -B_{11}^{-1}B_{12} \\ 0 & I & 0 \\ B_{21}B_{11}^{-1} & 0 & B_{22} - B_{21}B_{11}^{-1}B_{12} \end{bmatrix}. \tag{2.29}$$

The product of $H_1$ and $H_2$ is given by

$$H_2H_1 = \begin{bmatrix} B_{11} & B_{12}A_{21} & B_{12}A_{22} \\ 0 & A_{11} & A_{12} \\ B_{21} & B_{22}A_{21} & B_{22}A_{22} \end{bmatrix},$$

and by applying the exchange operator we find that

$$\operatorname{exc}(H_2H_1) = \begin{array}{c} \\ p \\ m-p \end{array} \begin{array}{c} \begin{array}{cc} p & m-p \end{array} \\ \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \end{array},$$

where

$$\begin{aligned} C_{11} &= \begin{bmatrix} B_{11}^{-1} & -B_{11}^{-1}B_{12}A_{21}A_{11}^{-1} \\ 0 & A_{11}^{-1} \end{bmatrix}, \\ C_{12} &= \begin{bmatrix} -B_{11}^{-1}B_{12}(A_{22} - A_{21}A_{11}^{-1}A_{12}) \\ -A_{11}^{-1}A_{12} \end{bmatrix}, \\ C_{21} &= \begin{bmatrix} B_{21}B_{11}^{-1} & (B_{22} - B_{21}B_{11}^{-1}B_{12})A_{21}A_{11}^{-1} \end{bmatrix}, \\ C_{22} &= (B_{22} - B_{21}B_{11}^{-1}B_{12})(A_{22} - A_{21}A_{11}^{-1}A_{12}). \end{aligned}$$

It is easily verified by multiplying (2.28) and (2.29) that

$$G_2G_1 = \begin{array}{c} \\ p \\ m-p \end{array} \begin{array}{c} \begin{array}{cc} p & m-p \end{array} \\ \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} \end{array} = \operatorname{exc}(H_2H_1).$$

It is also easy to verify that $C_{11}$ is nonsingular, and since the exchange operator is involutory, we have $\operatorname{exc}(G_2G_1) = H_2H_1$. □

The following lemma considers the errors involved when combining two nonoverlapping hyperbolic rotations.

**Lemma 2.2.11.** *Consider two hyperbolic transformations, $H_1 \in \mathbb{R}^{m\times m}$ agreeing with the identity matrix in rows and columns $1{:}\,t$ and $H_2 \in \mathbb{R}^{m\times m}$ agreeing with the identity matrix in rows and columns $t+1{:}\,p$, where $1 \le t \le p$. Suppose that the errors, $E_i$, in applying $H_1$ to the matrix $\begin{bmatrix} R^T & S^T & X^T \end{bmatrix}^T$, to give in*

*exact arithmetic* $[\, R^T \quad S_1^T \quad X_1^T \,]^T$, *are described by*

$$H_1 \begin{bmatrix} R \\ S \\ X + E_2 \end{bmatrix} = \begin{matrix} n \\ \begin{bmatrix} R \\ S_1 + E_1 \\ X_1 \end{bmatrix} \end{matrix} \begin{matrix} t \\ p-t \\ m-p \end{matrix} \tag{2.30}$$

*and the errors,* $F_i$, *in applying* $H_2$ *to the right hand side of* (2.30), *to give* $[\, R_1^T \quad (S_1 + E_1)^T \quad X_2^T \,]^T$ *in exact arithmetic, are described by*

$$H_2 \begin{bmatrix} R \\ S_1 + E_1 \\ X_1 + F_2 \end{bmatrix} = \begin{matrix} n \\ \begin{bmatrix} R_1 + F_1 \\ S_1 + E_1 \\ X_2 \end{bmatrix} \end{matrix} \begin{matrix} t \\ p-t \\ m-p \end{matrix} , \tag{2.31}$$

*where*

$$\max_{i=1,2} \|E_i\|_2 \le \mu \max(\|S_1\|_2, \|X\|_2),$$
$$\max_{i=1,2} \|F_i\|_2 \le \mu \max(\|R_1\|_2, \|X_1\|_2).$$

*Then*

$$H_2 H_1 \begin{bmatrix} R \\ S \\ X + \Delta X \end{bmatrix} = \begin{bmatrix} R_1 + \Delta R_1 \\ S_1 + \Delta S_1 \\ X_2 \end{bmatrix},$$

*where*

$$\max \left( \left\| \begin{bmatrix} \Delta R_1 \\ \Delta S_1 \end{bmatrix} \right\|_2, \|\Delta X\|_2 \right) \le 6\mu \max \left( \left\| \begin{bmatrix} R_1 \\ S_1 \end{bmatrix} \right\|_2, \|X\|_2 \right) + O(\mu^2).$$

*Proof.* First let us consider the hyperbolic transformations without the perturbations. Let $H_1$ and $H_2$ be two nonoverlapping hyperbolic transformations which satisfy

$$H_1 \begin{bmatrix} R \\ S \\ X \end{bmatrix} = \begin{bmatrix} R \\ S_1 \\ X_1 \end{bmatrix}, \quad H_2 \begin{bmatrix} R \\ S_1 \\ X_1 \end{bmatrix} = \begin{bmatrix} R_1 \\ S_1 \\ X_2 \end{bmatrix},$$

so that

$$H_2 H_1 \begin{bmatrix} R \\ S \\ X \end{bmatrix} = \begin{bmatrix} R_1 \\ S_1 \\ X_2 \end{bmatrix}.$$

Using the exchange operator, the two hyperbolic transformations can be rewritten in terms of orthogonal transformations $G_1$ and $G_2$ as

$$G_1 \begin{bmatrix} S_1 \\ X \end{bmatrix} = \begin{bmatrix} S \\ X_1 \end{bmatrix},$$
$$G_2 \begin{bmatrix} R_1 \\ X_1 \end{bmatrix} = \begin{bmatrix} R \\ X_2 \end{bmatrix},$$

where we express the relations in terms of the affected components only. These two relations can be rewritten as

$$\begin{bmatrix} R_1 \\ S_1 \\ X \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & G_1^T \end{bmatrix} \begin{bmatrix} R_1 \\ S \\ X_1 \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & G_1^T \end{bmatrix} \widetilde{G}_2^T \begin{bmatrix} R \\ S \\ X_2 \end{bmatrix} \equiv G \begin{bmatrix} R \\ S \\ X_2 \end{bmatrix}, \qquad (2.32)$$

where $\widetilde{G}_2([1\colon t, p+1\colon m], [1\colon t, p+1\colon m]) = G_2$ and elsewhere $\widetilde{G}_2$ agrees with the identity matrix, and $G$ is orthogonal.

Including the perturbations from (2.30) and (2.31) in the analysis, the perturbed version of (2.32) is

$$\begin{aligned} \begin{bmatrix} R_1 + F_1 \\ S_1 + E_1 \\ X + E_2 \end{bmatrix} &= \begin{bmatrix} I & 0 \\ 0 & G_1^T \end{bmatrix} \begin{bmatrix} R_1 + F_1 \\ S \\ X_1 \end{bmatrix} \\ &= \begin{bmatrix} I & 0 \\ 0 & G_1^T \end{bmatrix} \left( \widetilde{G}_2^T \begin{bmatrix} R \\ S \\ X_2 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ F_2 \end{bmatrix} \right). \qquad (2.33) \end{aligned}$$

This may be rewritten as

$$G_2 G_1 \begin{bmatrix} R_1 + \Delta R_1 \\ S_1 + \Delta S_1 \\ X + \Delta X \end{bmatrix} = \begin{bmatrix} R \\ S \\ X_2 \end{bmatrix}, \qquad (2.34)$$

where

$$\max\left(\left\|\begin{bmatrix}\Delta R_1\\ \Delta S_1\end{bmatrix}\right\|_2, \|\Delta X\|_2\right) \le 3\mu\max(\|R_1\|_2, \|S_1\|_2, \|X\|_2, \|X_1\|_2).$$

By renaming $E_1$ and $F_1$, and using Lemma 2.2.10 after applying the exchange operator, (2.34) can be written as

$$H_2H_1\begin{bmatrix}R\\ S\\ X+\Delta X\end{bmatrix} = \begin{bmatrix}R_1+\Delta R_1\\ S_1+\Delta S_1\\ X_2\end{bmatrix}.$$

We note that

$$\|X_1\|_2 \le 2\max(\|S_1\|_2, \|X\|_2) + O(\mu),$$

and therefore we have

$$\begin{aligned}\max\left(\left\|\begin{bmatrix}\Delta R_1\\ \Delta S_1\end{bmatrix}\right\|_2, \|\Delta X\|_2\right) &\le 6\mu\max(\|R_1\|_2, \|S_1\|_2, \|X\|_2) + O(\mu^2)\\ &\le 6\mu\max\left(\left\|\begin{bmatrix}R_1\\ S_1\end{bmatrix}\right\|_2, \|X\|_2\right) + O(\mu^2). \qquad \square\end{aligned}$$

This result is similar to that in [9] except that we have been able to write the error result in terms of the hyperbolic transformations $H_1$ and $H_2$ rather than in terms of $\mathrm{exc}(H_1)$ and $\mathrm{exc}(H_2)$ as in (2.34).

In [9] the analysis in the proof of Lemma 2.2.11 is used inductively $r-1$ times to give a result for the errors involved in applying a product of $r$ mutually nonoverlapping hyperbolic transformations $H_1$, $H_2$, ..., $H_r$. However, the precise details are not given. We choose not to use this approach but instead give a single more precise result for combining the errors of $r$ nonoverlapping hyperbolic rotations.

Consider hyperbolic transformations $H_i \in \mathbb{R}^{m\times m}$, for $i = 1\colon r$, that agree with the identity in all but rows and columns $t_{i-1}\colon t_i$ and $t_r\colon t_{r+1}$, where $t_0 = 1$, $t_r = p$, $t_{r+1} = m$ and $0 < t_1 < t_2 < \cdots < t_{r+1}$. The transformation $H_i$ is

applied to the matrix

$$A = \begin{matrix} n \\ \left[\begin{array}{c} A_1^{(1)} \\ A_2^{(1)} \\ \vdots \\ A_r^{(1)} \\ A_{r+1}^{(1)} \end{array}\right] \end{matrix} \begin{matrix} \\ t_1 \\ t_2 - t_1 \\ \vdots \\ t_r - t_{r-1} \\ t_{r+1} - t_r \end{matrix}$$

so that

$$\begin{bmatrix} A_i^{(2)} \\ A_{r+1}^{(i+1)} \end{bmatrix} = \widetilde{H}_i \begin{bmatrix} A_i^{(1)} \\ A_{r+1}^{(i)} \end{bmatrix},$$

where $\widetilde{H}_i = H_i([t_{i-1}\colon t_i, t_r\colon t_{r+1}], [t_{i-1}\colon t_i, t_r\colon t_{r+1}])$. After all the hyperbolic transformations have been applied we have

$$\begin{bmatrix} A_1^{(2)} \\ \vdots \\ A_r^{(2)} \\ A_{r+1}^{(r+1)} \end{bmatrix} = H_r \ldots H_2 H_1 \begin{bmatrix} A_1^{(1)} \\ \vdots \\ A_r^{(1)} \\ A_{r+1}^{(1)} \end{bmatrix}.$$

We consider the errors in combining $r$ hyperbolic transformations in the following lemma.

**Lemma 2.2.12.** *Consider hyperbolic transformations $H_i \in \mathbb{R}^{m\times m}$, $i = 1\colon r$, where $H_i$ agrees with the identity in all but rows and columns $t_{i-1}\colon t_i$ and $t_r\colon t_{r+1}$, where $t_0 = 1$, $t_r = p$, $t_{r+1} = m$ and $0 < t_1 < t_2 < \cdots < t_{r+1}$. Suppose that the errors $E_i$ and $D_i$, in applying $\widetilde{H}_i = H_i([t_{i-1}\colon t_i, t_r\colon t_{r+1}], [t_{i-1}\colon t_i, t_r\colon t_{r+1}])$ to the matrix $\begin{bmatrix} A_i^{(1)} \\ A_{r+1}^{(i)} \end{bmatrix}$ to give in exact arithmetic $\begin{bmatrix} A_i^{(2)} \\ A_{r+1}^{(i+1)} \end{bmatrix}$, are described by*

$$\widetilde{H}_i \begin{bmatrix} A_i^{(1)} \\ A_{r+1}^{(i)} + D_i \end{bmatrix} = \begin{matrix} n \\ \left[\begin{array}{c} A_i^{(2)} + E_i \\ A_{r+1}^{(i+1)} \end{array}\right] \end{matrix} \begin{matrix} \\ t_i - t_{i-1} \\ t_{r+1} - t_r \end{matrix}, \quad i = 1\colon r, \tag{2.35}$$

*where*

$$\max(\|D_i\|_2, \|E_i\|_2) \le \mu \max(\|A_{r+1}^{(i)}\|_2, \|A_i^{(2)}\|_2).$$

*Then*

$$H_r \dots H_2 H_1 \begin{bmatrix} A_1^{(1)} \\ \vdots \\ A_r^{(1)} \\ A_{r+1}^{(1)} + \Delta A_{r+1}^{(1)} \end{bmatrix} = \begin{bmatrix} A_1^{(2)} + \Delta A_1^{(2)} \\ \vdots \\ A_r^{(2)} + \Delta A_r^{(2)} \\ A_{r+1}^{(r+1)} \end{bmatrix},$$

*where*

$$\max(\|\Delta A_{r+1}^{(1)}\|_2, \|\Delta A_{1:r}^{(2)}\|_2) \leq (2r^2 - r)\mu \max(\|A_{r+1}^{(1)}\|_2, \|A_{1:r}^{(2)}\|_2) + O(\mu^2)$$

*and*

$$A_{1:r}^{(2)} = \begin{bmatrix} A_1^{(2)} \\ \vdots \\ A_r^{(2)} \end{bmatrix}, \quad \Delta A_{1:r}^{(2)} = \begin{bmatrix} \Delta A_1^{(2)} \\ \vdots \\ \Delta A_r^{(2)} \end{bmatrix}. \tag{2.36}$$

*Proof.* Using the exchange operator, the $r$ hyperbolic transformations, (2.35), can be rewritten in terms of orthogonal transformations as

$$\widetilde{G}_i \begin{bmatrix} A_i^{(2)} + E_i \\ A_{r+1}^{(i)} + D_i \end{bmatrix} = \begin{bmatrix} A_i^{(1)} \\ A_{r+1}^{(i+1)} \end{bmatrix}.$$

The $r$ orthogonal relations can be rewritten as

$$\begin{bmatrix} A_1^{(2)} + E_1 \\ \vdots \\ A_r^{(2)} + E_r \\ A_{r+1}^{(1)} + D_1 \end{bmatrix} = G_1^T \left( \cdots G_{r-1}^T \left( G_r^T \begin{bmatrix} A_1^{(1)} \\ \vdots \\ A_r^{(1)} \\ A_{r+1}^{(r+1)} \end{bmatrix} - \begin{bmatrix} 0 \\ \vdots \\ 0 \\ D_r \end{bmatrix} \right) - \cdots - \begin{bmatrix} 0 \\ \vdots \\ 0 \\ D_2 \end{bmatrix} \right)$$

$$= G_1^T G_2^T \dots G_r^T \begin{bmatrix} A_1^{(1)} \\ \vdots \\ A_r^{(1)} \\ A_{r+1}^{(r+1)} \end{bmatrix} - \begin{bmatrix} Y_1 \\ \vdots \\ Y_r \\ Y_{r+1} \end{bmatrix},$$

where $G_i([t_{i-1}\colon t_i, t_r\colon t_{r+1}], [t_{i-1}\colon t_i, t_r\colon t_{r+1}]) = \widetilde{G}_i$ and elsewhere $G_i$ agrees with the identity matrix, and

$$\|Y\|_2 \leq (r-1)\mu \max_{i=2:r}(\|A_{r+1}^{(i)}\|_2, \|A_i^{(2)}\|_2).$$

Using the inequality

$$\left\| \begin{bmatrix} E_1 \\ \vdots \\ E_r \end{bmatrix} \right\|_2 \leq \sum_{i=1}^{r} \|E_i\| \leq r\mu \max_{i=1:r}(\|A_{r+1}^{(i)}\|_2, \|A_i^{(2)}\|_2),$$

we can collect the error terms so that

$$G_r \ldots G_2 G_1 \begin{bmatrix} A_1^{(2)} + \Delta A_1^{(2)} \\ \vdots \\ A_r^{(2)} + \Delta A_r^{(2)} \\ A_{r+1}^{(1)} + \Delta A_{r+1}^{(1)} \end{bmatrix} = \begin{bmatrix} A_1^{(1)} \\ \vdots \\ A_r^{(1)} \\ A_{r+1}^{(r+1)} \end{bmatrix}, \tag{2.37}$$

where

$$\max(\|\Delta A_{r+1}^{(1)}\|_2, \|\Delta A_{1:r}^{(2)}\|_2) \leq (2r-1)\mu \max_{i=1:r}(\|A_{r+1}^{(i)}\|_2, \|A_i^{(2)}\|_2)$$

and $\Delta A_{1:r}^{(2)}$ is defined by (2.36).

Since

$$\begin{aligned} \|A_{r+1}^{(i+1)}\|_2 &\leq \|A_{r+1}^{(i)}\|_2 + \|A_i^{(2)}\|_2 + O(\mu) \\ &\leq \sum_{j=1}^{i} \|A_j^{(2)}\|_2 + \|A_{r+1}^{(1)}\|_2 + O(\mu), \end{aligned}$$

we have

$$\begin{aligned} \max(\|\Delta A_{r+1}^{(1)}\|_2, \|\Delta A_{1:r}^{(2)}\|_2) &\leq (2r^2 - r)\mu \max_{i=1:r}(\|A_{r+1}^{(1)}\|_2, \|A_i^{(2)}\|_2) + O(\mu^2) \\ &\leq (2r^2 - r)\mu \max(\|A_{r+1}^{(1)}\|_2, \|A_{1:r}^{(2)}\|_2) + O(\mu^2). \end{aligned}$$

Applying the exchange operator to (2.37) and using Lemma 2.2.10 inductively we obtain the desired result. □

The errors involved in applying a hyperbolic rotation using the representation H3 or H4 in mixed form, and by the OD procedure, described in Lemmas 2.2.6 and 2.2.8 respectively, satisfy (2.35). Therefore, by Lemma 2.2.12, providing the hyperbolic rotations are nonoverlapping it is possible to apply a sequence of them in a mixed forward-backward stable way.

In Section 2.5 we describe an algorithm for computing the hyperbolic QR factorization of a matrix using nonoverlapping hyperbolic transformations. We will make use of Lemma 2.2.12 when conducting a rounding error analysis of the algorithm.

### 2.2.6 Unified Rotations

In Section 2.2.1 we defined the hyperbolic rotation that zeroed the second component of $x \in \mathbb{C}^2$ to be

$$H = \begin{bmatrix} \bar{c} & -\bar{s} \\ -s & c \end{bmatrix},$$

where

$$c = \frac{x_1}{\sqrt{|x_1|^2 - |x_2|^2}}, \quad s = \frac{x_2}{\sqrt{|x_1|^2 - |x_2|^2}}, \tag{2.38}$$

if $|x_1| > |x_2|$. This is called a *Type 1* hyperbolic rotation and we showed it to be $J$-unitary in Section 2.2.1.

If we allow $|x_1| < |x_2|$ then we can obtain a form of hyperbolic rotation that gives real $c$ and $s$ for $x \in \mathbb{R}^2$ and satisfies

$$\begin{bmatrix} y_1 \\ 0 \end{bmatrix} = H \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

if $c$ and $s$ are defined by

$$c = \frac{x_1}{\sqrt{|x_2|^2 - |x_1|^2}}, \quad s = \frac{x_2}{\sqrt{|x_2|^2 - |x_1|^2}}. \tag{2.39}$$

This is called a *Type 2* hyperbolic rotation.

The matrix $H$ is now not $J$-unitary but satisfies

$$\begin{bmatrix} \bar{c} & -\bar{s} \\ -s & c \end{bmatrix}^* \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} \bar{c} & -\bar{s} \\ -s & c \end{bmatrix} = \begin{bmatrix} |c|^2 - |s|^2 & 0 \\ 0 & |s|^2 - |c|^2 \end{bmatrix}$$
$$= \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix},$$

and hence $H$ is $(J_1, J_2)$-unitary, where

$$J_1 = \pm \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

and $J_2 = -J_1$.

*Unified Rotations*, first introduced by Bojanczyk, Qiao and Steinhardt [11], include Givens and both types of hyperbolic rotations. Given a signature matrix $J_1 = \text{diag}(\sigma_1, \sigma_2)$, $\sigma_1 = \pm 1$, $\sigma_2 = \pm 1$, a unified rotation has the form

$$Q = \begin{bmatrix} \bar{c} & \frac{\sigma_2}{\sigma_1}\bar{s} \\ -s & c \end{bmatrix},$$

where $c$ and $s$ satisfy

$$\sigma_1 |c|^2 + \sigma_2 |s|^2 = \widetilde{\sigma}_1, \quad \widetilde{\sigma}_1 = \pm 1.$$

If we define $\widetilde{\sigma}_2 = \sigma_1 \sigma_2 \widetilde{\sigma}_1$ then $Q$ is $(J_1, J_2)$-unitary where $J_2 = \text{diag}(\widetilde{\sigma}_1, \widetilde{\sigma}_2)$.

If $\sigma_1 = \sigma_2$ then $Q$ is a Givens rotation and $c$ and $s$ are given by (1.5). If $\sigma_1 \neq \sigma_2$ then $Q$ is a hyperbolic rotation where $c$ and $s$ are given by (2.38) if $|x_1| > |x_2|$ and (2.39) if $|x_1| < |x_2|$. Unified rotations are not defined for $|x_1| = |x_2|$ when $\sigma_1 \neq \sigma_2$.

A unified rotation is $(J_1, J_2)$-unitary where

$$J_2 = \begin{cases} J_1 & \text{if } J_1 = \pm I, \\ J_1 & \text{if } J_1 \neq \pm I \text{ and } |x_1| > |x_2|, \\ -J_1 & \text{if } J_1 \neq \pm I \text{ and } |x_1| < |x_2|. \end{cases}$$

In Section 2.2.2 we showed there are various ways of representing and applying Type 1 hyperbolic rotations. We now present the analogous ways of representing and applying Type 2 hyperbolic rotations.

In Section 2.2.3 we showed that for a Type 1 hyperbolic rotation, $c$ and $s$ could be computed accurately by using the H3 and H4 representations. The analogous representations for a Type 2 hyperbolic rotation are

$$\text{H3}: \quad c = \frac{x_1}{(|x_2| + |x_1|)(|x_2| - |x_1|)}, \quad s = \frac{x_2}{(|x_2| + |x_1|)(|x_2| - |x_1|)} \tag{2.40}$$

and

$$\text{H4}: \quad c = \frac{x_1 s}{x_2}, \quad s = \frac{\text{sign}(x_2)}{\sqrt{2e - e^2}}, \tag{2.41}$$

where $e = (|x_2| - |x_1|)/|x_2|$. It not difficult to show that $c$ and $s$ computed using (2.40) or (2.41) satisfy similar error results to their Type 1 equivalent.

The following function computes $c$ and $s$, which define the unified rotation, according to H4 (2.7) for the Type 1 rotation and analogously for the Type 2 rotation.

**Algorithm 2.2.13.** *Given $x = [x_1, x_2]^T$ and $J_1$, $c$ and $s$ which define a unified rotation $Q$ are computed such that $Qx$ has zero second component and $Q$ is $(J_1, J_2)$-unitary.*

```
1  function [c, s, J_2] = urotate(x, J_1)
2  σ = J_1(1,1)J_1(2,2)
3  if |x_1| = −σ|x_2|
4       error - no unified rotation exists
5  end
6  J_2 = J_1
7  if |x_1| > |x_2|
8       if σ = 1
9            t = |x_2|/|x_1|, d = 1 + |t|^2
10           c = sign(x_1)/√d, s = tc
11      else
12           e = (|x_1| − |x_2|)/|x_1|
13           c = sign(x_1)/√(2e − e^2), s = (x_2/x_1)c
14      end
15 else
16      if σ = 1
17           t = |x_1|/|x_2|, d = 1 + |t|^2
18           s = sign(x_2)/√d, c = ts
19      else
20           J_2 = −J_2
21           e = (|x_2| − |x_1|)/|x_2|
22           s = sign(x_2)/√(2e − e^2), c = (x_1/x_2)s
23      end
```

24 end

A Type 2 hyperbolic rotation may be applied in a mixed form, similar to that for a Type 1 hyperbolic rotation. If we are applying $H$ to $a \in \mathbb{C}^2$ to obtain $b = Ha$, then this is achieved using

$$b_1 = \bar{c}a_1 - \bar{s}a_2,$$
$$b_2 = -\frac{c}{\bar{s}}b_1 - \frac{a_1}{\bar{s}}.$$

In Section 2.2.3 we showed that applying a Type 1 hyperbolic rotation in mixed form using the H3 or H4 representation is mixed forward-backward stable. The following lemma shows that a similar result is true for the Type 2 case.

**Lemma 2.2.14.** *Let a real Type 2 hyperbolic rotation, $H \in \mathbb{R}^{2\times 2}$, constructed using H3 (2.40) or H4 (2.41), be applied in mixed form. Let $a = [a_1 \ a_2]^T \in \mathbb{R}^2$ and $b = [b_1 \ b_2]^T = Ha$. Then the computed $\widehat{b}$ satisfies*

$$\widehat{b} + \Delta b = H(a + \Delta a),$$

*where* $\Delta b = \begin{bmatrix} \Delta b_1 \\ 0 \end{bmatrix}$, $\Delta a = \begin{bmatrix} \Delta a_1 \\ 0 \end{bmatrix}$ *and*

$$\text{H3}: \quad \max(|\Delta b_1|, |\Delta a_1|) \le \gamma_{20} \max(|\widehat{b}_1|, |a_1|),$$
$$\text{H4}: \quad \max(|\Delta b_1|, |\Delta a_1|) \le \gamma_{82} \max(|\widehat{b}_1|, |a_1|).$$

*Proof.* We will only prove the result for H4.

It is easy to show that

$$fl(c) = c(1+\eta_1), \quad |\eta_1| \le \gamma_{20},$$
$$fl(s) = s(1+\eta_2), \quad |\eta_2| \le \gamma_{18},$$

and therefore

$$\widehat{b}_1 = (c(1+\eta_1)a_1(1+\delta_1) - s(1+\eta_2)a_1(1+\delta_2))(1+\delta_3).$$

This can be rewritten as

$$a_2 = \frac{c}{s} a_1(1+\epsilon_1) - \frac{\widehat{b}_1}{s}(1+\epsilon_2), \tag{2.42}$$

where $|\epsilon_1| \leq \gamma_{40}$ and $|\epsilon_2| \leq \gamma_{40}$. The computed second component of $b$ satisfies

$$\widehat{b}_2 = -\frac{a_1}{s}(1+\epsilon_3) + \widehat{b}_1 \frac{c}{s}(1+\epsilon_4), \tag{2.43}$$

where $|\epsilon_3| \leq \gamma_{38}$ and $|\epsilon_4| \leq \gamma_{41}$.

Combining (2.42) and (2.43), we obtain

$$\begin{bmatrix} a_2 \\ \widehat{b}_2 \end{bmatrix} = (G + \Delta G) \begin{bmatrix} \widehat{b}_1 \\ a_1 \end{bmatrix}, \quad |\Delta G| \leq \gamma_{41} |G|,$$

where

$$G = \begin{bmatrix} \widetilde{c} & \widetilde{s} \\ -\widetilde{s} & \widetilde{c} \end{bmatrix}, \quad \widetilde{c} = -1/s, \ \widetilde{s} = -c/s,$$

is orthogonal. This can be rewritten as

$$\begin{bmatrix} a_2 \\ \widehat{b}_2 \end{bmatrix} = G \begin{bmatrix} \widehat{b}_1 + \Delta b_1 \\ a_1 + \Delta a_1 \end{bmatrix}, \tag{2.44}$$

with

$$\begin{bmatrix} \Delta b_1 \\ \Delta a_1 \end{bmatrix} = G^T \Delta G \begin{bmatrix} \widehat{b}_1 \\ a_1 \end{bmatrix}$$

and

$$\begin{aligned} \max(|\Delta b_1|, |\Delta a_1|) &\leq \gamma_{41}(1 + 2|\widetilde{c}||\widetilde{s}|) \max(|\widehat{b}_1|, |a_1|) \\ &\leq \gamma_{82} \max(|\widehat{b}_1|, |a_1|). \end{aligned}$$

Using the exchange operator we can rewrite (2.44) as

$$\begin{bmatrix} \widehat{b}_1 + \Delta b_1 \\ \widehat{b}_2 \end{bmatrix} = \begin{bmatrix} -s & c \\ c & -s \end{bmatrix} \begin{bmatrix} a_2 \\ a_1 + \Delta a_1 \end{bmatrix},$$

and hence we have

$$\begin{bmatrix} \widehat{b}_1 + \Delta b_1 \\ \widehat{b}_2 \end{bmatrix} = H \begin{bmatrix} a_1 + \Delta a_1 \\ a_2 \end{bmatrix}. \qquad \square$$

The following function applies the unified rotation directly if the rotation corresponds to a Givens rotation, and in mixed form if the rotation is a Type 1 or Type 2 hyperbolic rotation.

**Algorithm 2.2.15.** *Applies a unified rotation defined by $c$ and $s$ to a matrix $A \in \mathbb{C}^{2\times n}$. If the unified rotation corresponds to a hyperbolic rotation then it is applied in mixed form.*

```
1  function B =uapply(c, s, A, J_1, J_2)
2  σ = J_1(1,1)J_1(2,2)
3  B(1,:) = c̄A(1,:) + σs̄A(2,:)
4  if σ = 1
5        B(2,:) = −sA(1,:) + cA(2,:)
6  elseif J_1(1,1) = J_2(1,1)
7        B(2,:) = (−sB(1,:) + A(2,:))/c̄
8  else
9        B(2,:) = (−cB(1,:) − A(1,:))/s̄
10 end
```

It is also possible to apply a Type 2 hyperbolic rotation in an analogous way to the OD procedure for a Type 1 rotation, H5 (2.12). This is achieved by applying $H$ to $a \in \mathbb{R}^2$ to obtain $b = Ha$ using

$$b = G(D(Qa)), \tag{2.45}$$

where

$$G = \begin{bmatrix} -1 & 1 \\ 1 & 1 \end{bmatrix}, \quad D = \begin{bmatrix} d/2 & 0 \\ 0 & 1/(2d) \end{bmatrix}, \quad Q = \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}, \quad d = \sqrt{\frac{x_1 + x_2}{x_2 - x_1}}.$$

In Section 2.2.3 we showed that applying a Type 1 hyperbolic rotation using the OD procedure is mixed forward-backward stable. The following lemma shows that this is also true for the Type 2 case.

**Lemma 2.2.16.** *Let a Type 2 hyperbolic rotation, $H = QDQ^T$, be applied using* (2.45). *Let $a = [a_1\ a_2]^T \in \mathbb{R}^2$ and $b = [b_1\ b_2]^T = Ha$. Then the computed*

$\widehat{b}$ *satisfies*

$$\begin{bmatrix} \widehat{b}_1 + \Delta_1 \\ \widehat{b}_2 \end{bmatrix} = H \begin{bmatrix} a_1 + \Delta_2 \\ a_2 \end{bmatrix},$$

*where* $\max(|\Delta_1|, |\Delta_2|) \leq 14\gamma_6 \max(|\widehat{b}_1|, |a_1|)$.

*Proof.* The proof is similar to the arguments used in the proofs of Lemmas 2.2.7 and 2.2.8 and Corollary 2.2.9, except that we make use of the fact that if

$$\begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = H \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}$$

then

$$\begin{bmatrix} a_1 \\ b_2 \end{bmatrix} = G \begin{bmatrix} b_1 \\ a_1 \end{bmatrix},$$

where

$$G = \begin{bmatrix} -1/s & -c/s \\ c/s & -1/s \end{bmatrix}$$

is orthogonal. □

The following function describes how a unified rotation may be constructed and applied by combining the OD procedure representation, H5 (2.12), and its analogous Type 2 representation.

**Algorithm 2.2.17.** *Given* $x = [x_1 \ x_2]^T \in \mathbb{R}^2$ *and* $J_1 = \mathrm{diag}(\pm 1) \in \mathbb{R}^{2\times 2}$, *a unified rotation* $Q$ *that is* $(J_1, J_2)$ *orthogonal, such that* $Qx$ *has zero second component, is applied to a matrix* $A \in \mathbb{R}^{2\times n}$ *to obtain* $B \in \mathbb{R}^{2\times n}$. *If the unified rotation corresponds to a hyperbolic rotation then it is applied using the OD procedure.*

```
1  function [B, J2] = uapply_od(x, A, J1)
2  σ = J1(1,1)J1(2,2)
3  if σ = 1
4       if |x1| > |x2|
5            t = |x2|/|x1|, d = 1 + |t|^2
6            c = sign(x1)/√d, s = tc
7       else
```

8 $\quad\quad\quad t = |x_1|/|x_2|,\ d = 1 + |t|^2$

9 $\quad\quad\quad s = \text{sign}(x_2)/\sqrt{d},\ c = ts$

10 $\quad$ end

11 $\quad J_2 = J_1$

12 $\quad B(1,:) = \bar{c}A(1,:) + \sigma\bar{s}A(2,:),\ B(2,:) = -sA(1,:) + cA(2,:)$

13 else

14 $\quad$ if $|x_1| > |x_2|$, $\alpha = 1$, else, $\alpha = -1$, end

15 $\quad d = \sqrt{\frac{x_1+x_2}{\alpha(x_1-x_2)}}$

16 $\quad J_2 = \alpha J_1$

17 $\quad \widetilde{B}(1,:) = A(1,:) - A(2,:),\ \widetilde{B}(2,:) = A(1,:) + A(2,:)$

18 $\quad \widetilde{B}(1,:) = (d/2)\widetilde{B}(1,:),\ \widetilde{B}(1,:) = (1/(2d))\widetilde{B}(2,:)$

19 $\quad B(1,:) = \alpha\widetilde{B}(1,:) + \widetilde{B}(2,:),\ B(2,:) = -\alpha\widetilde{B}(1,:) + \widetilde{B}(2,:)$

20 end

The functions described in this section are implemented as MATLAB functions in Appendix A.1.

## 2.3 Fast Rotations

In this section we first consider fast Givens rotations in order to develop fast hyperbolic rotations. We restrict our attention to the real case for simplicity.

### 2.3.1 Fast Givens Rotations

Suppose a Givens rotation, $G$, is to be applied to a matrix $A$ so that $B = GA$, where $G$ is equal to the identity except for

$$G([p,q],[p,q]) = \begin{bmatrix} c & -s \\ s & c \end{bmatrix}, \quad c^2 + s^2 = 1.$$

The motivation for a fast rotation is to reduce the number of multiplications needed to apply a rotation, by expressing the matrix $A$ in the form $A = D\widetilde{A}$, where $D = \text{diag}(d_1, \ldots, d_m)$ and $\widetilde{A}$ is scaled accordingly. When a sequence of rotations are applied to $A = D\widetilde{A}$ then $\widetilde{A}$ and $D$ are updated separately and the

product of the two factors is calculated after all rotations have been applied. The matrix $D$ is initially set to the identity matrix.

If $A = D\widetilde{A}$ then applying a Givens rotation can be represented as

$$B = GA = GD\widetilde{A} = KF\widetilde{A} = K\widetilde{B},$$

where $K = \mathrm{diag}(k_1, \ldots, k_m)$ and $F$ is defined as

$$F = \begin{bmatrix} I_{p-1} & 0 & 0 & 0 & 0 \\ 0 & f_{pp} & 0 & f_{pq} & 0 \\ 0 & 0 & I_{q-p-1} & 0 & 0 \\ 0 & f_{qp} & 0 & f_{qq} & 0 \\ 0 & 0 & 0 & 0 & I_{n-q} \end{bmatrix}. \tag{2.46}$$

We call $F$ a *fast Givens rotation* if the choices of $f_{pp}, f_{pq}, f_{qp}$ and $f_{qq}$ result in reducing the number of multiplications in applying a Givens rotation.

There are several ways of choosing $F$ and $K$ so that the number of multiplications is reduced. Two common choices are of the form

$$F_{pq} = \begin{bmatrix} f_{pp} & f_{pq} \\ f_{qp} & f_{qq} \end{bmatrix} = \begin{bmatrix} 1 & \beta \\ \alpha & 1 \end{bmatrix}$$

or

$$F_{pq} = \begin{bmatrix} f_{pp} & f_{pq} \\ f_{qp} & f_{qq} \end{bmatrix} = \begin{bmatrix} \beta & 1 \\ -1 & \alpha \end{bmatrix}.$$

Suppose we wish to zero the second component of $[\widetilde{x}_{pj}\ \widetilde{x}_{qj}]^T$, where $X = D\widetilde{X}$, by a fast rotation. This will affect the $p$th and $q$th rows of the matrix which the fast rotation is being applied to, say $\widetilde{A}$, where $A = D\widetilde{A}$. The following formula applies the fast rotation corresponding to the first choice of $F$ to $\widetilde{A}$ to give $\widetilde{B}$, where $B = K\widetilde{B}$ and $K$ is a diagonal matrix:

$$\widetilde{B}(p,:) = \widetilde{A}(p,:) + \beta\widetilde{A}(q,:), \tag{2.47a}$$

$$\widetilde{B}(q,:) = \alpha\widetilde{A}(p,:) + \widetilde{A}(q,:), \tag{2.47b}$$

where $\alpha = -t, \beta = t\frac{d_q^2}{d_p^2}$ and $t = \widetilde{x}_{qj}/\widetilde{x}_{pj}$, and

$$c = \frac{d_p \widetilde{x}_{pj}}{\sqrt{d_p^2 \widetilde{x}_{pj}^2 + d_q^2 \widetilde{x}_{qj}^2}}, \quad k_p = cd_p, \quad k_q = cd_q.$$

A problem with fast rotations is that underflow can occur if $D$ is multiplied by $|c| \ll 1$. The risk of underflow can be reduced by using the following alternative formula, corresponding to the second choice of $F$, when $c < 1/2$:

$$\widetilde{B}(p,:) = \beta \widetilde{A}(p,:) + \widetilde{A}(q,:), \tag{2.48a}$$

$$\widetilde{B}(q,:) = -\widetilde{A}(p,:) + \alpha \widetilde{A}(q,:), \tag{2.48b}$$

where $\alpha = \frac{1}{t}, \beta = \frac{1}{t}\frac{d_p^2}{d_q^2}$ and $t = \widetilde{x}_{qj}/\widetilde{x}_{pj}$, and

$$s = \frac{d_q \widetilde{x}_{qj}}{\sqrt{d_p^2 \widetilde{x}_{pj}^2 + d_q^2 \widetilde{x}_{qj}^2}}, \quad k_p = sd_q, \quad k_q = sd_p.$$

By an appropriate choice of the forms of $F$, the decrease in magnitude of each element of $D$ can be bounded by $1/\sqrt{2}$. The risk of underflow in $D$ can be further reduced through the use of self scaling fast rotations [2].

To avoid overflow we may compute $c$ and $s$ using

$$c = \frac{1}{\sqrt{1+v}}, \quad \text{where } v = \left(\frac{d_q \widetilde{x}_{qj}}{d_p \widetilde{x}_{pj}}\right)^2,$$

and

$$s = \frac{1}{\sqrt{1+w}}, \quad \text{where } w = \left(\frac{d_p \widetilde{x}_{pj}}{d_q \widetilde{x}_{qj}}\right)^2.$$

### 2.3.2 Fast Hyperbolic Rotations

In a similar way that hyperbolic rotations are analogous to Givens rotations, *fast hyperbolic rotations* are analogous to fast Givens rotations. These have been considered in [46] for computing the hyperbolic singular value decomposition of a matrix. However, the fast hyperbolic rotations in [46] are applied in

two different ways, of which one is unstable. We detail the methods of applying a fast hyperbolic rotation and give a rounding error analysis and numerical experiments to determine the stability properties.

We recall that a hyperbolic rotation, $H$, is equal to the identity except for

$$H([p,q],[p,q]) = \begin{bmatrix} c & -s \\ -s & c \end{bmatrix}, \quad c^2 - s^2 = 1.$$

Applying a hyperbolic rotation to $A = D\widetilde{A}$ can be represented as

$$B = HA = HD\widetilde{A} = KF\widetilde{A} = K\widetilde{B},$$

where $F$ is of the form (2.46) and $K = \mathrm{diag}(k_1, \ldots, k_m)$.

We now give two sets of equations for fast hyperbolic rotations which are analogous to equations (2.47) and (2.48). Firstly we have

$$\widetilde{B}(p,:) = \widetilde{A}(p,:) + \beta\widetilde{A}(q,:), \tag{2.49a}$$

$$\widetilde{B}(q,:) = \alpha\widetilde{A}(p,:) + \widetilde{A}(q,:), \tag{2.49b}$$

where $\alpha = -t, \beta = -t\frac{d_p^2}{d_q^2}$ and $t = \widetilde{x}_{qj}/\widetilde{x}_{pj}$, and

$$c = \frac{d_p\widetilde{x}_{pj}}{\sqrt{(d_p\widetilde{x}_{pj})^2 - (d_q\widetilde{x}_{qj})^2}}, \quad k_p = cd_p, \quad k_q = cd_q.$$

Secondly we have

$$\widetilde{B}(p,:) = \beta\widetilde{A}(p,:) - \widetilde{A}(q,:), \tag{2.50a}$$

$$\widetilde{B}(q,:) = -\widetilde{A}(p,:) + \alpha\widetilde{A}(q,:), \tag{2.50b}$$

where $\alpha = \frac{1}{t}, \beta = \frac{1}{t}\frac{d_p^2}{d_q^2}$ and $t = \widetilde{x}_{qj}/\widetilde{x}_{pj}$, and

$$s = \frac{d_q\widetilde{x}_{qj}}{\sqrt{(d_p\widetilde{x}_{pj})^2 - (d_q\widetilde{x}_{qj})^2}}, \quad k_p = sd_q, \quad k_q = sd_p.$$

We note that to obtain real $c$ and $s$ we must impose the condition $d_p\widetilde{x}_{pj} > d_q\widetilde{x}_{qj}$, and that this corresponds to $x_{pj} > x_{qj}$, which must be satisfied for standard hyperbolic rotations.

Unlike for hyperbolic rotations, we know of no stable method of computing $c$ and $s$ in floating point arithmetic for fast hyperbolic rotations. But, in order to conduct a rounding error analysis of applying fast hyperbolic rotations we make the assumption that $c$ and $s$ can be computed exactly.

The fast hyperbolic rotations (2.49) and (2.50) are applied directly to the matrix $\widetilde{A}$. For hyperbolic rotations it has been shown in Section 2.2.3 that for stability reasons, the hyperbolic rotation should be applied in mixed form rather than directly. The following lemma considers the error in applying a fast hyperbolic rotation directly.

**Lemma 2.3.1.** *Suppose $a' = Da$ where $a', a \in \mathbb{R}^2$ and $D \in \mathbb{R}^{2\times 2}$ is diagonal. Let a fast hyperbolic rotation, $F$, that is equivalent to applying the hyperbolic rotation $H$ to $a'$ to give $b' = Ha'$, be applied to the vector $a$ using* (2.49) *or* (2.50) *so that $b = Fa$ and $b' = Kb$, where $K \in \mathbb{R}^{2\times 2}$ is diagonal. Then the errors in computing $b$ and $K$ satisfy*

$$\widehat{K}\widehat{b} = H(Da + \Delta), \quad |\Delta| \leq \gamma_8 |H|^2 |D||a|.$$

*Proof.* We give the proof for the fast hyperbolic rotation given by (2.49). The other case is similar. We make the assumption that $c$ and $s$ can be computed exactly so that $fl(c) = c$ and $fl(s) = s$.

The computed scalars $\widehat{\alpha}$ and $\widehat{\beta}$ satisfy

$$\widehat{\alpha} = \alpha(1 + \theta_1), \tag{2.51}$$

$$\widehat{\beta} = \beta(1 + \theta_5), \tag{2.52}$$

and the computed diagonal components of $K = \mathrm{diag}(k_1, k_2)$ satisfy

$$\widehat{k}_1 = cd_1(1 + \theta_1'),$$

$$\widehat{k}_2 = cd_2(1 + \theta_1'').$$

Using (2.52) we find that

$$\begin{aligned}\widehat{b_1} &= (a_1 + \beta(1+\theta_5)a_2(1+\delta_1))(1+\delta_2) \\ &= a_1(1+\theta_1''') + \beta a_2(1+\theta_7),\end{aligned}$$

and using (2.51) we have

$$\begin{aligned}\widehat{b}_2 &= (\alpha(1+\theta_1)a_1(1+\delta_3) + a_2)(1+\delta_4) \\ &= \alpha a_1(1+\theta_3) + a_2(1+\theta_1'''').\end{aligned}$$

Multiplying $\widehat{K}$ by $\widehat{b}$ in exact arithmetic gives

$$\begin{aligned}\widehat{k}_1 b_1 &= cd_1(1+\theta_1')(a_1(1+\theta_1''') + \beta a_2(1+\theta_7)) \\ &= cd_1a_1(1+\theta_2) - sd_2a_2(1+\theta_8),\end{aligned} \tag{2.53}$$

and

$$\begin{aligned}\widehat{k}_2 b_2 &= cd_1(1+\theta_1'')(\alpha a_1(1+\theta_3) + a_2(1+\theta_1'''')) \\ &= -sd_1a_1(1+\theta_4) + cd_2a_2(1+\theta_2').\end{aligned} \tag{2.54}$$

Combining (2.53) and (2.54) we obtain

$$\begin{aligned}Kb &= \begin{bmatrix} c(1+\theta_2) & -s(1+\theta_8) \\ -s(1+\theta_4) & c(1+\theta_2') \end{bmatrix} Da \\ &= H\left( Da + H^{-1} \begin{bmatrix} c\theta_2 & -s\theta_8 \\ -s\theta_4 & c\theta_2' \end{bmatrix} Da \right) \\ &= H(Da + \Delta),\end{aligned}$$

where $|\Delta| \leq \gamma_8 |H|^2 |D||a|$. □

For fast hyperbolic rotations applied directly we have been unable to find error bounds that do not depend on $H$. In Section 2.2.3 we showed that hyperbolic rotations applied directly are unstable, and therefore we would like a form of fast hyperbolic rotation that is applied in a manner similar to applying a hyperbolic rotation in mixed form.

We can obtain a fast hyperbolic rotation that is applied in a mixed way if we consider chained fast rotations, which are of the form

$$F_{pq} = \begin{bmatrix} 1 & 0 \\ -\alpha & 1 \end{bmatrix} \begin{bmatrix} 1 & -\beta \\ 0 & 1 \end{bmatrix}, \tag{2.55}$$

$$F_{pq} = \begin{bmatrix} 1 & -\beta \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -\alpha & 1 \end{bmatrix}. \tag{2.56}$$

The rotation of the form (2.55) is applied to a vector $x \in \mathbb{R}^2$ by

$$y_1 = x_1 - \beta x_2,$$

$$y_2 = x_2 - \alpha y_1.$$

We note that the calculation of $y_2$ uses the already calculated $y_1$ and is therefore in a mixed form. The rotation (2.56) is applied in a similar way.

The formula for applying the first fast hyperbolic rotation in mixed form is given by

$$\widetilde{B}(p,:) = \widetilde{A}(p,:) - \beta \widetilde{A}(q,:), \tag{2.57a}$$

$$\widetilde{B}(q,:) = \widetilde{A}(q,:) - \alpha \widetilde{B}(p,:), \tag{2.57b}$$

where, $\alpha = d_p sc/d_q$, $\beta = (d_q s)/(d_p c)$ and

$$c = \frac{d_p \widetilde{x}_{pj}}{\sqrt{(d_p \widetilde{x}_{pj})^2 - (d_q \widetilde{x}_{qj})^2}}, \quad s = \frac{d_q \widetilde{x}_{qj}}{\sqrt{(d_p \widetilde{x}_{pj})^2 - (d_q \widetilde{x}_{qj})^2}}, \quad k_p = d_p c, \quad k_q = \frac{d_q}{c}. \tag{2.58}$$

Secondly we have

$$\widetilde{B}(q,:) = \widetilde{A}(q,:) - \alpha \widetilde{A}(p,:), \tag{2.59a}$$

$$\widetilde{B}(p,:) = \widetilde{A}(p,:) - \beta \widetilde{B}(q,:), \tag{2.59b}$$

where, $\alpha = (s d_p)/(c d_q)$, $\beta = cs d_q/d_p$ and

$$c = \frac{d_p \widetilde{x}_{pj}}{\sqrt{(d_p \widetilde{x}_{pj})^2 - (d_q \widetilde{x}_{qj})^2}}, \quad s = \frac{d_q \widetilde{x}_{qj}}{\sqrt{(d_p \widetilde{x}_{pj})^2 - (d_q \widetilde{x}_{qj})^2}}, \quad k_p = \frac{d_p}{c}, \quad k_q = d_q c.$$

The following lemma shows that the fast hyperbolic rotations applied by (2.57) and (2.59) are mixed forward-backward stable.

**Lemma 2.3.2.** *Suppose $a' = Da$ where $a', a \in \mathbb{R}^2$ and $D \in \mathbb{R}^{2\times 2}$ is diagonal. Let a fast hyperbolic rotation, $F$, that is equivalent to applying the hyperbolic rotation $H$ to $a'$ to give $b' = Ha'$, be applied to the vector $a$ using (2.57) or (2.59) so that $b = Fa$ and $b' = Kb$, where $K \in \mathbb{R}^{2\times 2}$ is diagonal. Then the errors in computing $b$ and $K$ satisfy*

$$\widehat{K}\widehat{b} + \begin{bmatrix} \Delta_1 \\ 0 \end{bmatrix} = H\left( Da + \begin{bmatrix} 0 \\ \Delta_2 \end{bmatrix} \right),$$

*where* $\max(|\Delta_1|, |\Delta_2|) \leq \gamma_{12} \max(|\widehat{k}_1\widehat{b}_1|, |d_2 a_2|)$.

*Proof.* We give the proof for the fast hyperbolic rotation given by (2.57). The other case is proved in a similar way. We make the assumption that $c$ and $s$ can be computed exactly so that $fl(c) = c$ and $fl(s) = s$.

Let us consider the computed scalars $\widehat{\beta}$, $\widehat{\alpha}$ and the diagonal elements of $\widehat{K} = \mathrm{diag}(\widehat{k}_1, \widehat{k}_2)$. Firstly

$$\widehat{\beta} = \beta(1 + \theta_5), \tag{2.60}$$

and

$$\widehat{k}_1 = d_1 c(1 + \theta_1).$$

Alternatively the error in computing $\widehat{k}_1$ can be expressed as

$$d_1 = \frac{\widehat{k}_1}{c}(1 + \theta_1').$$

The computed version of the scalar $\alpha = (d_1 sc)/d_2 = k_1 s/d_2$ satisfies

$$\widehat{\alpha} = \frac{\widehat{k}_1 s}{d_2}(1 + \theta_3), \tag{2.61}$$

and finally

$$\widehat{k}_2 = \frac{d_2}{c}(1 + \theta_1'').$$

Using (2.60) we have that

$$\begin{aligned} \widehat{b}_1 &= (a_1 - \widehat{\beta} a_2(1 + \delta_2))(1 + \delta_3) \\ &= (a_1 - \beta(1 + \theta_5) a_2(1 + \delta_2))(1 + \delta_3), \end{aligned}$$

which may be rearranged so that

$$a_1 = \widehat{b}_1(1+\theta_2) + \beta a_2(1+\theta_6).$$

Also, using (2.61) we have

$$\begin{aligned}\hat{b}_2 &= (-\widehat{\alpha} b_1(1+\delta_4) + a_2)(1+\delta_5) \\ &= -\frac{\widehat{k}_1}{d_2} s b_1(1+\theta_5') + a_2(1+\theta_1''').\end{aligned}$$

Multiplying $d_1$ by $a_1$ and $\widehat{k}_2$ by $\widehat{b}_2$ in exact arithmetic gives

$$\begin{aligned}d_1 a_1 &= \frac{\widehat{k}_1}{c}(1+\theta_1')\widehat{b}_1(1+\theta_2) + d_1\beta a_2(1+\theta_6) \\ &= \frac{\widehat{k}_1}{c}\widehat{b}_1(1+\theta_3) + d_2\frac{s}{c}a_2(1+\theta_6), \qquad (2.62)\end{aligned}$$

and

$$\begin{aligned}\widehat{k}_2\widehat{b}_2 &= \frac{d_2}{c}(1+\theta_1'')\left(-\frac{\widehat{k}_1}{d_2}b_1(1+\theta_5') + a_2(1+\theta_1''')\right) \\ &= -\widehat{k}_1\frac{s}{c}\widehat{b}_1(1+\theta_6') + \frac{d_2}{c}a_2(1+\theta_2'). \qquad (2.63)\end{aligned}$$

Combining (2.62) and (2.63) we have

$$\begin{bmatrix} d_1 a_1 \\ \widehat{k}_2\widehat{b}_2 \end{bmatrix} = (G + \Delta G)\begin{bmatrix} \widehat{k}_1\widehat{b}_1 \\ d_2 a_1 \end{bmatrix},$$

where

$$G = \begin{bmatrix} 1/c & s/c \\ -s/c & c \end{bmatrix}$$

is orthogonal and $|\Delta| \leq \gamma_6$. This result can be rewritten as

$$\begin{bmatrix} d_1 a_1 \\ \widehat{k}_2\widehat{b}_2 \end{bmatrix} = G\begin{bmatrix} \widehat{k}_1\widehat{b}_1 + \Delta_1 \\ d_2 a_2 + \Delta_2 \end{bmatrix},$$

where

$$\begin{bmatrix} \Delta_1 \\ \Delta_2 \end{bmatrix} = G^T \Delta G \begin{bmatrix} \widehat{k}_1\widehat{b}_1 \\ d_2 a_2 \end{bmatrix},$$

so that

$$\begin{aligned}\max(|\Delta_1|, |\Delta_2|) &\leq \gamma_6(1 + 2|\widetilde{c}|, |\widetilde{s}|) \max(|\widehat{k}_1\widehat{b}_1|, |d_2a_2|) \\ &= \gamma_{12} \max(|\widehat{k}_1\widehat{b}_1|, |d_2a_2|),\end{aligned}$$

where $\widetilde{c} = a/c$ and $\widetilde{s} = s/c$. Finally, applying the exchange operator gives

$$\begin{bmatrix} \widehat{k}_1\widehat{b}_1 + \Delta_1 \\ \widehat{k}_2\widehat{b}_2 \end{bmatrix} = H \begin{bmatrix} d_1a_1 \\ d_2a_2 + \Delta_2 \end{bmatrix}. \qquad \square$$

Numerical experiments similar to those in Section 2.2.4 can be used to show the advantage of applying a fast hyperbolic rotation in a mixed way.

If a method of applying a fast hyperbolic rotation is mixed forward-backward stable then there should exist a hyperbolic rotation $\widetilde{H} \in \mathbb{R}^{2\times 2}$ such that

$$\begin{bmatrix} \widehat{k}_1\widehat{b}_1 + \Delta_1 \\ \widehat{k}_2\widehat{b}_2 \end{bmatrix} = \widetilde{H} \begin{bmatrix} d_1a_1 \\ d_2a_2 + \Delta_2 \end{bmatrix}$$

with $\|[\Delta_1 \ \Delta_2]^T\|_2$ close to

$$\delta = u \left\| \begin{bmatrix} \widehat{k}_1\widehat{b}_1 \\ d_2a_2 \end{bmatrix} \right\|_2, \tag{2.64}$$

where $u$ is the unit roundoff. Using a similar argument to that in Section 2.2.4 we can show that the defining variable of the hyperbolic rotation $\widetilde{H}$ that gives the minimum value of $\|[\Delta_1 \ \Delta_2]^T\|_2$ can be found by solving the constrained least squares problem

$$\min_y \left( \left\| \begin{bmatrix} d_1a_1 & -\widehat{k}_2\widehat{b}_2 \\ \widehat{k}_2\widehat{b}_2 & d_1a_1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} - \begin{bmatrix} \widehat{k}_1\widehat{b}_1 \\ d_2a_2 \end{bmatrix} \right\| \right) \quad \text{subject to } \|y\|_2 = 1. \tag{2.65}$$

Let $H \in \mathbb{R}^{2\times 2}$ be the hyperbolic rotation such that $Hx$ has zero second component. For various $x \in \mathbb{R}^2$, $a \in \mathbb{R}^2$ and diagonal $D \in \mathbb{R}^{2\times 2}$ we computed $b \in \mathbb{R}^2$ and the diagonal $K \in \mathbb{R}^{2\times 2}$ such that $Kb = HDa$ using (2.49) and (2.57). We then solved (2.65) in 100-digit arithmetic using MATLAB's Symbolic Math Toolbox.

We recall that we would expect the value of (2.65) to be close to (2.64) if the method of applying the fast hyperbolic rotation is stable. Therefore the results in Table 2.4 show that applying the fast hyperbolic rotation by (2.49) is unstable. Further experiments show that using (2.50) is also unstable. However, all our experiments of applying a fast hyperbolic rotation using (2.57) or (2.59) have behaved in a mixed forward-backward stable way. We note that this is despite the fact that we have used an unstable method to compute $c$ and $s$, which is highlighted by the quantity $\|H - fl(H)\|_2/\|H\|_2$ in Table 2.4.

Since our numerical experiments suggest that computing $c$ and $s$ using (2.58) does not affect the stability when applying fast hyperbolic rotation in mixed form, and using (2.49) or (2.50) is unstable, we recommend the use of (2.57) or (2.59).

Table 2.4: The value (2.65) computed for a fast hyperbolic rotation applied directly (2.49) and in a mixed form (2.57), with $\widetilde{x} = [1, 1-\alpha]^T$, $d = [10, 10-\beta]^T$ and $a = [5, 5-\theta]^T$. The value $\delta$ is given by (2.64). The errors $\|H - fl(H)\|_2/\|H\|_2$ in forming $H$ using (2.58) are also shown.

| $\alpha$ | $\beta$ | $\theta$ | $\delta$ | (2.49) | (2.57) | $\frac{\|H-fl(H)\|_2}{\|H\|_2}$ |
|---|---|---|---|---|---|---|
| 1.0e-04 | 1.0e-08 | 1.0e-01 | 9.6e-15 | 9.5e-14 | 1.1e-14 | 3.2e-13 |
| 1.0e-06 | 1.0e-04 | 1.0e-02 | 6.0e-15 | 1.9e-13 | 4.7e-15 | 4.1e-12 |
| 1.0e-10 | 1.0e-08 | 1.0e-02 | 2.4e-13 | 2.1e-10 | 1.2e-13 | 4.5e-11 |
| 1.0e-10 | 1.0e-08 | 1.0e-06 | 5.6e-15 | 5.1e-13 | 3.3e-15 | 4.5e-11 |
| 1.0e-12 | 1.0e-08 | 1.0e-04 | 6.1e-15 | 5.7e-11 | 2.7e-16 | 3.3e-08 |
| 1.0e-12 | 1.0e-12 | 1.0e-02 | 7.5e-12 | 2.9e-09 | 1.1e-11 | 3.0e-05 |

In Section 2.3.1 we showed how to reduce the risk of underflow when using fast Givens rotations. For fast hyperbolic rotations we note that $c$ can be arbitrarily large and hence underflow and overflow is possible for $k_p$ and $k_q$ in (2.57) and (2.59). In order to minimise this risk we use (2.57) if $|d_p| \leq |d_q|$ and (2.59) otherwise.

The following function computes the defining variables of the fast hyperbolic

rotation according to the procedure described above.

**Algorithm 2.3.3.** *Computes the $\alpha, \beta \in \mathbb{R}$ and $k \in \mathbb{R}^2$ which define the fast hyperbolic rotation that zeros the second component of $[d_1x_1 \;\; d_2x_2]^T \in \mathbb{R}^2$ . The output variable "type" is the form of fast hyperbolic rotation used and determines how it is applied.*

```
1  [α, β, k, type] = hrotate_fast(x, d)
2  if |d1x1| ≤ |d2x2|
3        error - Fast hyperbolic rotation not defined
4  end
5  e = (|d1x1| − |d2x2|)/|d1x1|
6  c = 1/√(2e − e²)
7  s = d2x2c/(d1x1)
8  if |d1| ≥ |d2|
9        α = d1cs/d2
10       β = d2s/(d1c)
11       k1 = cd1, k2 = d2/c
12       type = 1
13 else
14       α = sd1/(cd2)
15       β = csd2/d1
16       k1 = d1/c, k2 = d2c
17       type = 2
18 end
```

The following function applies the fast hyperbolic rotation constructed according to Algorithm 2.3.3, to a $2 \times n$ matrix.

**Algorithm 2.3.4.** *Applies the fast hyperbolic rotation, defined by $\alpha$ and $\beta$, to the matrix $A \in \mathbb{R}^{2\times n}$. The value "type" is determined in Algorithm 2.3.3.*

```
1  B = happly_fast(A, α, β, type)
2  if type = 1
3        B(1, :) = A(1, :) − βA(2, :)
4        B(2, :) = A(2, :) − αB(1, :)
```

```
5  else
6        B(2,:) = A(2,:) − αA(1,:)
7        B(1,:) = A(1,:) − βB(2,:)
8  end
```

These functions have been implemented for use in MATLAB as the functions `hrotate_fast.m` and `happly_fast.m` and are given in Appendix A.1. Also given in Appendix A.1 are the equivalent functions for constructing and applying fast Givens rotations.

## 2.4 Hyperbolic Householder Transformations

A *hyperbolic Householder transformation* is a $J$-unitary transformation that zeros all but one element of a vector. Given a signature matrix, $J = \mathrm{diag}(\pm 1)$, a hyperbolic Householder transformation has the form

$$H = J - \frac{2vv^*}{v^*Jv}, \quad v^*Jv \neq 0.$$

For a hyperbolic Householder transformation we have that

$$\begin{aligned} H^*JH &= \left(J - \frac{2vv^*}{v^*Jv}\right)^* J \left(J - \frac{2vv^*}{v^*Jv}\right) \\ &= J^3 - 4\frac{vv^*}{v^*Jv} + \frac{4v(v^*Jv)v^*}{(v^*Jv)^2} \\ &= J, \end{aligned}$$

and so $H$ is $J$-unitary.

A hyperbolic Householder transformation may be applied to a vector $x \in \mathbb{C}^n$ so that all but one element of $x$ is zeroed,

$$Hx = \beta e_i. \tag{2.66}$$

We define

$$\mathrm{sign}(x_i) = \begin{cases} \frac{x_i}{|x_i|}, & \text{if } x_i \neq 0 \\ 1 & \text{otherwise.} \end{cases}$$

**Theorem 2.4.1.** *If there is to exist a hyperbolic Householder transformation, $H$, such that $Hx = \beta e_i$ then $x$ and $J$ must satisfy*

$$x^*Jx \neq 0 \quad \textit{and} \quad \text{sign}(x^*Jx) = \text{sign}(J(i,i)). \tag{2.67}$$

*Proof.* Using (2.66) and the fact that $H$ is $J$-unitary we have that

$$x^*Jx = x^*H^*JHx = |\beta|^2 e_i^T J e_i = |\beta|^2 J(i,i)$$

and hence the result. □

If we assume (2.67) to be true and let

$$v = Jx + \sigma \text{sign}(x_i)|x^*Jx|^{1/2} e_i,$$

where $\sigma = \text{sign}(x^*Jx)$, then it is easy to verify that

$$v^*Jv = 2\sigma(|x^*Jx| + |x_i||x^*Jx|^{1/2})$$

and

$$v^*x = \sigma(|x^*Jx| + |x_i||x^*Jx|^{1/2})$$

so that

$$Hx = J - \frac{2v^*x}{v^*Jv}v = -\sigma \text{sign}(x_i)|x^*Jx|^{1/2} e_i.$$

The following function computes the hyperbolic Householder vector $v$ which defines $H$ such that $Hx$ produces a vector of zeros except for the $i$th component.

**Algorithm 2.4.2.** *Computes the hyperbolic Householder vector, $v$, which defines a hyperbolic Householder transformation, $Q$, such that $Qx$ has all but the $i$th element equal to zero.*

```
1  function v = hhouse(x, J, i)
2  if x*Jx = 0
3        error - no hyperbolic Householder transformation exists
4  end
```

```
5  if sign(x*Jx) = sign(J(i,i))
6        σ = sign(J(i,i))
7        v = Jx
8        v_i = v_i + σ sign(x_i) sqrt(|x*Jx|)
9  else
10       error - no hyperbolic Householder transformation exists
11 end
```

The structure of a hyperbolic Householder transformation may be exploited when applying it to a matrix $A \in \mathbb{C}^{m\times n}$, just as described for a Householder transformation in Section 1.6.2. For a hyperbolic Householder transformation we have

$$B = (J - \beta vv^*)A = JA - vw^*, \tag{2.68}$$

where $w = \beta A^*v$ and $\beta = 1/(v^*Jv)$. Applying the transformation using (2.68) reduces the cost by an order of magnitude compared with forming $H$ and multiplying with $A$.

Unlike ordinary Householder transformations, the hyperbolic versions are ill-conditioned. In [11] it is shown that the singular values of $H$ satisfy

$$\sigma_{\min}^{-1}(H) = \sigma_{\max}^{-1}(H) = \frac{v^*v}{|v^*Jv|} + \sqrt{\left(\frac{v^*v}{v^*Jv}\right)^2 - 1}.$$

For $J \neq I$ the ratio $v^*v/(v^*Jv)$ can be arbitrarily large and hence the condition number $\kappa_2(H) = \sigma_{\max}/\sigma_{\min}$ can also be large and the transformation ill-conditioned.

The $(J_1, J_2)$-orthogonal equivalent of hyperbolic Householder transformations are unified hyperbolic Householder transformations. For more details see [11].

A MATLAB implementation of Algorithm 2.4.2 is given by the function `hhouse.m` in Appendix A.1. The unified Householder transformation equivalent is given by the function `uhouse.m`.

As well as being able to apply a hyperbolic Householder transformation directly using (2.68), it is also possible to apply it in a mixed way by making use of the orthogonal matrix $\mathrm{exc}(H)$. It is claimed by Stewart and Stewart [52] that both these methods are mixed forward-backward stable although it is unclear how $v^*Jv$ may be computed in a stable way. Since we know of no stable method for computing $v^*Jv$ we recommend avoiding the use of hyperbolic Householder transformations, and in Section 2.5.3 suggest an alternative method of zeroing all but one element of a vector, due to Bojanczyk, Higham and Patel [9]. This alternative method has the advantage that we can show it to be stable. Also the condition number of this alternative method has been shown by Tisseur [53, Thm. 4.3] to be less than or equal to the condition number of the equivalent single hyperbolic Householder transformation.

## 2.5 Hyperbolic QR Factorization

### 2.5.1 Existence of the HR Factorization and Hyperbolic QR Factorization

In order to study the existence of the hyperbolic QR factorization we first prove a new theorem for the existence of the HR factorization. We extend a theorem of Bunse-Gerstner [12], which gives conditions for the existence of the HR factorization for $n \times n$ matrices, to deal with $m \times n$ matrices. We denote by $\mathrm{diag}_p^m(\pm 1)$ the set of $m \times m$ matrices with $p$ 1s and $(m-p)$ $-1$s on the diagonal, and zeros elsewhere. Given $A \in \mathbb{C}^{m\times n}$ and $J \in \mathrm{diag}_p^m(\pm 1)$ we say that there is an HR factorization of $A$ if $A = HR$ where $H$ is $(J, \widetilde{J})$-unitary, $\widetilde{J} \in \mathrm{diag}_p^m(\pm 1)$ and $R \in \mathbb{C}^{m\times n}$ is upper trapezoidal. The following theorem describes necessary conditions and sufficient conditions for the existence of an HR factorization.

**Theorem 2.5.1.** *Let $A \in \mathbb{C}^{m\times n}$ be of full rank and $J \in \operatorname{diag}_p^m(\pm 1)$ with $m \geq n$. Then there exists a $(J, \widetilde{J})$-unitary matrix $Q \in \mathbb{C}^{m\times m}$, for some $\widetilde{J} \in \operatorname{diag}_p^m(\pm 1)$, and an upper trapezoidal $R \in \mathbb{C}^{m\times n}$, such that $A = QR$, if all the leading principal minors of $A^*JA$ are nonsingular.*

*If the factorization exists then the product of the first $i$ diagonal elements of $\widetilde{J}$ is equal to the sign of the $i$th principal minor of $A^*JA$ for all $i \leq n$.*

*Proof.* Assume that all the leading principal minors of $A^*JA$ are nonsingular. Then, since $A^*JA$ is Hermitian, it can be factorized as

$$A^*JA = U^*DU,$$

where $U \in \mathbb{C}^{n\times n}$ is unit upper triangular and $D \in \mathbb{C}^{n\times n}$ is diagonal [22]. Clearly the product of the first $i$ elements of $D$ equals the $i$th principal leading minor of $A^*JA$. If we take $\widetilde{J}_1 = \operatorname{sign}(D)$ then

$$A^*JA = U^*|D|^{1/2}\widetilde{J}_1|D|^{1/2}U,$$

and the product of the first $i$ elements of $\widetilde{J}_1$ equals the sign of the $i$th principal leading minor of $A^*JA$.

We define $R_1 = |D|^{1/2}U$ and $Q_1 = AR_1^{-1}$. Since $A = Q_1|D|^{1/2}U$,

$$\begin{aligned} Q_1^*JQ_1 &= |D|^{-1/2}U^{-*}A^*JAU^{-1}|D|^{-1/2} \\ &= |D|^{-1/2}D|D|^{-1/2} \\ &= \widetilde{J}_1, \end{aligned}$$

where $\widetilde{J}_1 = \operatorname{diag}(\pm 1)$.

We now show that we can find $Q_2$ such that

$$[\,Q_1 \quad Q_2\,]^* \begin{bmatrix} J_1 & 0 \\ 0 & J_2 \end{bmatrix} [\,Q_1 \quad Q_2\,] = \begin{bmatrix} \widetilde{J}_1 & 0 \\ 0 & \widetilde{J}_2 \end{bmatrix},$$

for some $\widetilde{J}_2 \in \operatorname{diag}(\pm 1)$. Recall that if $Q \in \mathbb{C}^{m\times m}$ is $(J, \widetilde{J})$-unitary then

$$q_i^*(Jq_j) = \begin{cases} \pm 1 & i = j, \\ 0 & i \neq j. \end{cases}$$

It is easy to see that $\text{rank}(R_1) = n$ and hence $\text{rank}(Q_1) = n$. This means that the columns of $Q_1$ are linearly independent. We can therefore find $m - n$ linearly independent vectors that lie in the null space of $Q_1^* J \in \mathbb{C}^{n \times m}$. We can choose the columns of $Q_2$ to be these vectors, which can be transformed by $Q_2 \leftarrow Q_2 T$ so that $Q_2^* J Q_2 = \widetilde{J}_2$, where $\widetilde{J}_2 = \text{diag}(\{-1, 0, 1\})$.

Since $Q = [Q_1 \ Q_2]$ is nonsingular, by Sylvester's law of inertia (see, for example, [22]) we have that $\widetilde{J}$ has the same number of 1s and $-1$s as $J$, and therefore $Q$ is $(J, \widetilde{J})$-unitary.

Conversely, if the hyperbolic QR factorization exists then

$$\begin{aligned} A^* J A &= (QR)^* J Q R \\ &= R^* Q^* J Q R \\ &= R^* \widetilde{J} R, \end{aligned}$$

and therefore the product of the first $i$ diagonal elements of $\widetilde{J}$ is equal to the sign of the $i$th principal minor of $A^* J A$ for $i \leq n$. □

In the case where $A$ is rank deficient, a condition for existence of an $HR$ factorization is given by the following corollary.

**Corollary 2.5.2.** *Let $A \in \mathbb{C}^{m \times n}$ with $\text{rank}(A) = k < n$ and $J \in \text{diag}_p^m(\pm 1)$ with $m \geq n$. Then there exists a $(J, \widetilde{J})$-unitary matrix $Q \in \mathbb{C}^{m \times m}$, where $\widetilde{J} \in \text{diag}_p^m(\pm 1)$, and an upper triangular $R_{11} \in \mathbb{C}^{k \times k}$ such that*

$$B := AP = QR = Q \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix},$$

*where $P$ is a permutation matrix that swaps the columns of $A$ so that the first $k$ columns are linearly independent, if all the leading principal minors of*

$$B(:, 1\!:\!k)^* J B(:, 1\!:\!k) \tag{2.69}$$

*are nonzero.*

*Proof.* If all leading principal minors of (2.69) are nonzero then by Theorem 2.5.1 there exists a $(J, \widetilde{J})$-unitary $Q \in \mathbb{C}^{m\times m}$ and an upper triangular $R_{11} \in \mathbb{C}^{k\times k}$ such that

$$B(:, 1\colon k) = Q \begin{bmatrix} R_{11} \\ 0 \end{bmatrix}.$$

We can choose the columns of a matrix $R_{12} \in \mathbb{C}^{(m-k)\times n}$ to be

$$R_{12}(:, j) = Q^* B(:, k+j), \quad j = 1\colon m-k.$$

Then $B = QR$, where

$$R = \begin{bmatrix} R_{11} & R_{12} \\ 0 & 0 \end{bmatrix}. \qquad \square$$

We can now consider the hyperbolic QR factorization since this is a special case of the $HR$ factorization. Given $J = \operatorname{diag}(I_p, -I_q)$, $p + q = m$ and $A \in \mathbb{C}^{m\times n}$, $A$ has a hyperbolic QR factorization if we can write $A = QR$, where $Q$ is $J$-unitary and $R$ is upper trapezoidal. We obtain a result for the hyperbolic QR factorization by requiring $J = \widetilde{J}$ in Theorem 2.5.1.

**Theorem 2.5.3.** *Let $A \in \mathbb{C}^{m\times n}$ be of full rank, $J = \operatorname{diag}(I_p, -I_q)$, $p + q = m$, and $m \geq n$. There exists a $J$-unitary $Q \in \mathbb{C}^{m\times m}$ and an upper trapezoidal $R \in \mathbb{C}^{m\times n}$ such that $A = QR$ if the product of the first $i$ diagonal elements of $J$ is equal to the sign of the $i$th leading principal minor of $A^*JA$ for $i = 1\colon n$.*

*Proof.* If the product of the first $i$ diagonal elements of $J$ is equal to the sign of the $i$th principal minor of $A^*JA$, then by Theorem 2.5.1 there exists a $(J, \widetilde{J})$-unitary $Q$ and upper trapezoidal $R$ such that $A = QR$ and $J(1\colon n, 1\colon n) = \widetilde{J}(1\colon n, 1\colon n)$. Since the columns $q_{n+1}, \ldots, q_m$ of $Q$ can be permuted into any order and $\operatorname{inertia}(J) = \operatorname{inertia}(\widetilde{J})$, there is a column permutation of $Q$, say $\widetilde{Q}$, such that $A = \widetilde{Q}R$ and $\widetilde{Q}^* J \widetilde{Q} = \widetilde{J}$, where $J = \widetilde{J}$. $\square$

If $p \geq n$ then, for $i \leq n$, the first $i$ diagonal elements of $J$ are equal to 1, and by Theorem 2.5.3, there exists a hyperbolic QR factorization if the leading

principal minors of $A^*JA$ are positive. If $A^*JA$ is positive definite then this condition is satisfied, which gives us the following corollary which was also obtained in [9].

**Corollary 2.5.4.** *Let $A \in \mathbb{C}^{m\times n}$ be of full rank, $J = \mathrm{diag}(I_p, -I_q)$, $p+q = m$, $m \geq n$ and $p \geq n$. If $A^*JA$ is positive definite, there exists a $J$-unitary $Q \in \mathbb{C}^{m\times m}$ and upper trapezoidal $R \in \mathbb{C}^{m\times n}$ such that $A = QR$.*

### 2.5.2 Hyperbolic QR Using Hyperbolic Householder Transformations

The hyperbolic QR factorization can be computed using hyperbolic Householder transformations in an analogous way to how the QR factorization is computed by orthogonal Householder transformations. The matrix $A \in \mathbb{C}^{m\times n}$ is reduced to upper trapezoidal form by applying a sequence of hyperbolic Householder transformations to zero $A^{(k)}(k+1\colon m, k)$, where $A^{(k)}$ is the matrix $A$ after $k-1$ hyperbolic Householder transformations and $A^{(1)} = A$.

This gives the upper trapezoidal $R$ as

$$R = H_t \ldots H_2 H_1 A = QA,$$

where $H_i$ is the $i$th transformation and $t = \min(m-1, n)$. Since $H_1, H_2, \ldots, H_t$ are all $J$-unitary and a product of $J$-unitary matrices is also $J$-unitary, $Q$ is $J$-unitary. The method is summarised by the following function.

**Algorithm 2.5.5.** *Computes the hyperbolic QR factorization of an $m \times n$ matrix $A$. The matrix $Q$ is $J$-unitary, where $J = \mathrm{diag}(I_p, -I_{m-p})$, and $R$ is upper trapezoidal. Hyperbolic Householder transformations are used.*

```
1 function [Q, R] =hhqr(A, p)
2 Q = I_m
3 for i = 1:min(m − 1, n)
```

```
4       v =hhouse(A(i:m, i), J(i:m, i:m))
5       α = 2/(v*J(i:m, i:m)v)
6       A(i:m, i:n) = J(i:m, i:m)A(i:m, i:n) − αv(v*A(i:m, i:n))
7       Q(i:m, :) = J(i:m, i:m)Q(i:m, :) − αv(v*Q(i:m, :))
8   end
9   R = A
```

As noted in Section 2.4, we do not recommend using hyperbolic Householder transformations, as no stable representation is known. Instead we recommend using the procedure described in the following section.

### 2.5.3 Hyperbolic QR Using Hyperbolic Rotations

We now consider using hyperbolic rotations to compute the hyperbolic factorization of

$$A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \begin{matrix} p \\ q \end{matrix}, \quad p+q = m.$$

The rows of the submatrix $A_1$ corresponds to $I_p$ in $J$ and therefore unitary transformations can be used to reduce $A_1$ to upper trapezoidal form. We can then use hyperbolic transformations to reduce $A_2$ so that $A$ is upper trapezoidal. We do so using nonoverlapping hyperbolic transformations. The algorithm described is the same as that in [9], except that we do not impose the restriction $p \geq n$.

If $p \geq n$ then after $A_1$ has been reduced to upper triangular form we have

$$\begin{bmatrix} Q_1 & 0 \\ 0 & I_q \end{bmatrix} \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} = \begin{bmatrix} R_1 \\ 0 \\ A_2 \end{bmatrix}, \tag{2.70}$$

where $Q_1$ is unitary and $R_1 \in \mathbb{C}^{n \times n}$ is upper triangular. We use the notation

$$A^{(i)} = \begin{bmatrix} R_1^{(i)} \\ 0 \\ A_2^{(i)} \end{bmatrix}$$

to represent the right hand side of (2.70) after transformations have been applied so that the first $i-1$ columns of $A_2$ have been annihilated.

To annihilate the $i$th column of $A_2$ a Householder transformation is applied to the last $q$ rows of $A^{(i)}$, to give $A^{(i')}$, to remove all but the first element of $A^{(i)}(p+1, i\colon m)$. We note that the part of $J$ corresponding to $A_2$ is $-I_q$ and hence the unitary transformation is $J$-unitary. The element $A^{(i)}(i, p+1)$ is then removed by a hyperbolic rotation in the $(i, p+1)$ plane which is also $J$-unitary. Therefore we require $n$ Householder transformations and only $n$ hyperbolic rotations to annihilate $A_2$.

In Section 2.2.3 we showed that applying the hyperbolic rotation in mixed form or by the OD procedure is mixed forward-backward stable whereas no satisfactory stability result for applying the hyperbolic rotation directly is known. Therefore we apply the hyperbolic rotation either in mixed form or by the OD procedure.

After $A_2$ has been annihilated by a sequence of $J$-unitary transformations we have $QA = R$, where $Q$ is $J$-unitary and $R$ is upper trapezoidal.

In the case where $p < n$, $A_1$ is transformed so that it is upper trapezoidal and

$$\begin{bmatrix} Q_1 & 0 \\ 0 & I_q \end{bmatrix} \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} = \begin{bmatrix} \overset{p}{R_{11}} & \overset{n-p}{R_{12}} \\ A_{21} & A_{22} \end{bmatrix} \begin{matrix} p \\ q \end{matrix},$$

where $Q_1$ is unitary and $R_{11}$ is upper triangular. The submatrix $A_{21}$ is annihilated by the method described to annihilate $A_2$ in (2.70).

The submatrix $A_{22}$ must now be reduced to upper trapezoidal form. Since the rows of the submatrix $A_{22}$ correspond to $-I_q$ in $J$, we can use unitary transformations, as we did for $A_1$. The algorithm is summarised in the following function.

**Algorithm 2.5.6.** *Computes the hyperbolic QR factorization of* $A \in \mathbb{R}^{m\times n}$.

*The matrix $Q$ is $J$-unitary, where $J = \mathrm{diag}(I_p, -I_{m-p})$, and $R$ is upper trapezoidal. Householder transformations and at most $n$ hyperbolic rotations applied in mixed form are used.*

```
1  function [Q, R] = hqr(A, p)
2  Q = I_m
3  Compute QR factorization of A(1:p, :) so that Q'A(1:p, :) = R'
4  Q(1:p, 1:p) = Q'
5  A(1:p, :) = R'
6  for j = 1:min(m - 1, n)
7       Compute Householder transformation H_j such
              that H_j A(p + 1:m, j) = σe_1
8       A(p + 1:m, j:n) = H_j A(p + 1:m, j:n)
9       Q(p + 1:m, :) = H_j Q(p + 1:m, :)
10      [c, s] = hrotate(A([j, p + 1], j))
11      A([j, p + 1], j:n) = happly(c, s, A([j, p + 1], j:n))
12      Q([j, p + 1], :) = happly(c, s, Q([j, p + 1], :))
13 end
14 if p < n
15      Compute QR factorization Q'A(p + 1:m, p + 1:n) = R'
16      Q(p + 1:m, :) = Q'Q(p + 1:m, :)
17      A(p + 1:m, p + 1:n) = R'
18 end
19 R = A
```

Householder transformations are well known to be stable and we have shown in Section 2.2.3 that nonoverlapping hyperbolic rotations can be applied in a stable way. Therefore we would expect the method described by Algorithm 2.5.6 to be more stable than Algorithm 2.5.5, since we know of no stable way to apply hyperbolic Householder transformations. In Section 2.5.4 we prove that Algorithm 2.5.6 is conditionally backward stable and therefore suggest using this function instead of Algorithm 2.5.5.

### 2.5.4 Error Analysis and Numerical Experiments

We now consider the rounding error analysis for computing the hyperbolic QR factorization, using nonoverlapping hyperbolic rotations, of

$$A = \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \begin{matrix} p \\ q \end{matrix} \in \mathbb{R}^{m\times n}, \quad \overset{n}{}$$

so that $QA = R$, where $Q$ is $J$-orthogonal, $R$ is upper trapezoidal and $J = \mathrm{diag}(I_p, -I_q)$. We will assume that the hyperbolic rotations are constructed using either H3 or H4 and applied using the mixed form or by the OD procedure. We restrict our analysis to the case where $p \geq n$.

We first compute the QR factorization $A_1 = Q_1\widetilde{U}_1$, where $\widetilde{U}_1 = [\widetilde{R}_1^T\ 0]^T \in \mathbb{R}^{p\times n}$ is upper trapezoidal. The computed $\widetilde{U}_1$ is the exact factor of $A_1 + \Delta_1$, where $\|\Delta_1\|_F \leq \widetilde{\gamma}_{pn}\|A_1\|_F$ [30, Thm. 19.4]. We assume that $A_1$ is already upper trapezoidal and add the error $\Delta_1$ later.

Consider the $j$th column of $A$,

$$\begin{bmatrix} a_{1,j}^{(0)} \\ a_{2,j}^{(0)} \end{bmatrix} = \begin{bmatrix} A_1(:,j) \\ A_2(:,j) \end{bmatrix} \begin{matrix} p \\ q \end{matrix}. \quad \overset{n}{}$$

In the hyperbolic QR algorithm this column undergoes $n$ Householder transformations in the last $q$ components, and $n$ hyperbolic rotations in the planes $(1, p+1), \ldots, (n, p+1)$.

The first Householder transformation, $P_1$, agrees with the identity matrix in rows and columns $1\colon p$ and the first hyperbolic rotation, $H_1$, agrees with the identity matrix in rows and columns $2\colon p$ and $p+2\colon m$. Hence, (2.30) and (2.31) are satisfied with $E_1 = 0$ and $\mu = \widetilde{\gamma}_q$. We therefore have a particularly simple application of Lemma 2.2.11: (2.33) is satisfied with $E_1 = 0$, $E_2 = 0$, $F_1 = 0$ and $G_1^T$ is equal to the identity in rows and columns $1\colon p-1$. We can collect the

error terms and apply the exchange operator to the right hand side of (2.33) to find that

$$H_1 P_1 \begin{bmatrix} a_{1,j}^{(0)} \\ a_{2,j}^{(0)} + \Delta a_{2,j}^{(0)} \end{bmatrix} = \begin{bmatrix} a_{1,j}^{(1)} + \Delta a_{1,j}^{(1)} \\ a_{2,j}^{(1)} \end{bmatrix},$$

where

$$\max(\|\Delta a_{2,j}^{(0)}\|_2, \|\Delta a_{1,j}^{(1)}(1)\|_2) \leq \widetilde{\gamma}_q \max(\|a_{2,j}^{(0)}\|_2, \|a_{1,j}^{(1)}(1)\|_2).$$

We have that $\Delta a_{1,j}^{(1)}(2{:}\,p) = 0$ since rows $2{:}\,p$ are unaffected by the transformations.

Now consider any pair $H_k P_k$ where the Householder transformation $P_k$ agrees with the identity in rows and columns $1{:}\,p$, and the hyperbolic rotation $H_k$ agrees with the identity in all but rows and columns $k$ and $p+1$. If this pair of transformations is applied to the vector $[\alpha_1^T \ \alpha_2^T]^T$ to obtain $[\beta_1^T \ \beta_2^T]^T$, we have

$$H_k P_k \begin{bmatrix} \alpha_1 \\ \alpha_2 + \Delta\alpha_2 \end{bmatrix} = \begin{bmatrix} \beta_1 + \Delta\beta_2 \\ \beta_2 \end{bmatrix},$$

where

$$\max(\|\Delta\alpha_2\|_2, \|\Delta\beta_1(k)\|_2) \leq \widetilde{\gamma}_q \max(\|\alpha_2\|_2, \|\beta_1(k)\|_2).$$

We have that $\Delta\alpha_2(1{:}\,k-1)$ and $\Delta\alpha_2(k+1{:}\,p)$ are equal to zero since rows $1{:}\,k-1$ and $k+1{:}\,p$ are unaffected by the transformations.

The $H_i P_i$, for $i = 1{:}\,n$, are mutually nonoverlapping and hence we can use Lemma 2.2.12 to obtain

$$H_n P_n \ldots H_1 P_1 \begin{bmatrix} a_{1,j}^{(0)} \\ a_{2,j}^{(0)} + \Delta a_{2,j}^{(0)} \end{bmatrix} = \begin{bmatrix} a_{1,j}^{(j)} + \Delta a_{1,j}^{(j)} \\ a_{2,j}^{(j)} \end{bmatrix},$$

where

$$\max(\|\Delta a_{2,j}^{(0)}\|_2, \|\Delta a_{1,j}^{(j)}(1{:}\,j)\|_2) \leq \widetilde{\gamma}_{qj} \max(\|a_{2,j}^{(0)}\|_2, \|a_{1,j}^{(j)}(1{:}\,j)\|_2),$$

since only $j$ pairs of Householder transformations and hyperbolic rotations are applied to $a_j^{(0)}$ and $a_{1,j}^{(l)}(j+1{:}\,n) = 0$ for $l = 1{:}\,n$.

We note that $a_{1,j}^{(j)}(n\colon p) = 0$ and $a_{2,j}^{(j)} = 0$, hence

$$Q \begin{bmatrix} a_{1,j}^{(0)} \\ a_{2,j}^{(0)} + \Delta_2 \end{bmatrix} = \begin{matrix} n \\ \\ p-n \\ \\ q \end{matrix} \begin{bmatrix} \widehat{r}_j + \Delta_3 \\ 0 \\ 0 \end{bmatrix}, \tag{2.71}$$

where

$$\max_{i=2,3} \|\Delta_i\|_2 \leq \widetilde{\gamma}_{qj}, \max(\|a_{2,j}^{(0)}\|_2, \|\widehat{r}_j\|_2),$$

$\widehat{r}_j = \widehat{R}_1(:, j)$ and $Q$ is $J$-orthogonal. Applying the exchange operator to (2.71) gives

$$G \begin{bmatrix} \widehat{r}_j + \Delta_3 \\ 0 \\ a_{2,j}^{(0)} + \Delta_2 \end{bmatrix} = \begin{bmatrix} a_{1,j}^{(0)} \\ 0 \end{bmatrix},$$

where $G$ is orthogonal, and therefore

$$\|\widehat{r}_j\|_2 \leq \|a_{1,j}^{(0)}\|_2 + O(\widetilde{\gamma}_{qj}),$$
$$\|a_{2,j}^{(0)}\|_2 \leq \|a_{1,j}^{(0)}\|_2 + O(\widetilde{\gamma}_{qj}).$$

We thus have

$$\max_{i=2,3} \|\Delta_i\|_2 \leq \widetilde{\gamma}_{qj} \|a_{1,j}^{(0)}\|_2 + O(\widetilde{\gamma}_{qj}^2).$$

Putting these equations together for $j = 1\colon n$ gives

$$G \begin{bmatrix} \widehat{R}_1 + \overline{\Delta}_3 \\ 0 \\ A_2 + \overline{\Delta}_2 \end{bmatrix} = \begin{bmatrix} A_1 \\ 0 \end{bmatrix},$$

where

$$\|\overline{\Delta_i}\|_F \leq \widetilde{\gamma}_{qn} \|A_1\|_F, \quad i = 2, 3.$$

Applying the exchange operator and including the error, $\Delta_1$, for the initial QR factorization of $A_1$ gives

$$\overline{Q} \begin{bmatrix} A_1 + \Delta_1 \\ A_2 + \overline{\Delta}_2 \end{bmatrix} = \begin{bmatrix} \widehat{R}_1 + \overline{\Delta}_3 \\ 0 \\ 0 \end{bmatrix},$$

where $\overline{Q}$ is $J$-orthogonal and $\|\Delta_1\|_F \leq \widetilde{\gamma}_{pn}\|A_1\|_F$. This result is summarised in the following theorem.

**Theorem 2.5.7.** *Let $\widehat{R}_1 \in \mathbb{R}^{p\times n}$, $p \geq n$, be the computed upper triangular hyperbolic QR factor of $A \in \mathbb{R}^{m\times n}$ obtained by a combination of Householder transformations and nonoverlapping hyperbolic rotations applied in mixed form or by the OD procedure. Then there exists a $J$-orthogonal $Q \in \mathbb{R}^{m\times m}$ such that*

$$\begin{array}{c} \\ p \\ q \end{array} \begin{array}{c} n \\ \begin{bmatrix} A_1 + \Delta A_1 \\ A_2 + \Delta A_2 \end{bmatrix} \end{array} = Q \begin{bmatrix} \widehat{R}_1 + \Delta R_1 \\ 0 \end{bmatrix}, \tag{2.72}$$

*where*

$$\max(\|\Delta A_1\|_F, \|\Delta A_2\|_F, \|\Delta R_1\|_F) \leq \widetilde{\gamma}_{mn}\|A\|_F,$$

*and $J = \operatorname{diag}(I_p, -I_q)$.*

This result is a mixed forward-backward error result, which can be combined with the following lemma [51, pp. 302–304] to obtain a backward error result.

**Lemma 2.5.8.** *Let $m = p + q$ and $n \geq p$. Given a full rank matrix $A \in \mathbb{R}^{p\times n}$ and $E \in \mathbb{R}^{q\times n}$ there exists an orthogonal $Q \in \mathbb{R}^{m\times m}$ such that*

$$Q \begin{bmatrix} A \\ E \end{bmatrix} = \begin{bmatrix} A + F \\ 0 \end{bmatrix},$$

*where, for small $\|E\|_2$,*

$$\|F\|_2 \leq \frac{\|E\|_2^2}{2\sigma_{\min}(A)} + O(\|E\|_2^4).$$

After applying the exchange operator to (2.72) we can rewrite it as

$$\begin{bmatrix} \widehat{R}_1 \\ 0 \\ A_2 + \Delta A_2 \end{bmatrix} = G' \begin{bmatrix} A_1 + \Delta A_1 + \widetilde{\Delta}_1 \\ \widetilde{\Delta}_2 \end{bmatrix}, \quad \widetilde{\Delta} = -G'^T \begin{bmatrix} \Delta R_1 \\ 0 \end{bmatrix}, \tag{2.73}$$

where $G'$ is orthogonal. Applying Lemma 2.5.8 to the right hand side of (2.73), we have

$$\begin{bmatrix} \widehat{R}_1 \\ 0 \\ A_2 + \Delta A_2 \end{bmatrix} = \overline{G} \begin{bmatrix} A_1 + \overline{\Delta A}_1 \\ 0 \end{bmatrix}, \tag{2.74}$$

where $\overline{G}$ is orthogonal and

$$\begin{aligned} \|\Delta A_2\|_F &\le \widetilde{\gamma}_{mn} \|A\|_F, \\ \|\overline{\Delta A}_1\|_F &\le \frac{\widetilde{\gamma}_{mn}^2 \|A_1\|_F^2}{2\sigma_{\min}(A_1 + \Delta A_1 + \widetilde{\Delta}_1)} + O(u^4) \\ &\le \frac{\sqrt{n}}{2} (\kappa_2(A_1)\widetilde{\gamma}_{mn})\widetilde{\gamma}_{mn} \|A\|_F + O(u^3). \end{aligned}$$

Providing $\overline{G}(1\colon p, 1\colon p)$ is nonsingular we can rewrite (2.74) as

$$\begin{bmatrix} A_1 + \overline{\Delta A}_1 \\ A_2 + \Delta A_2 \end{bmatrix} = \widetilde{Q} \begin{bmatrix} \widehat{R}_1 \\ 0 \\ 0 \end{bmatrix},$$

where $\widetilde{Q}$ is $J$-orthogonal, and conclude that the hyperbolic QR factorization is backward stable if $\kappa_2(A_1)u$ is of order 1.

The result is summarised in the following theorem.

**Theorem 2.5.9.** *Let $\widehat{R}_1 \in \mathbb{R}^{p\times n}$, $p \ge n$, be the computed upper triangular hyperbolic QR factor of $A \in \mathbb{R}^{m\times n}$ obtained by a combination of Householder transformations and nonoverlapping hyperbolic rotations applied in mixed form or by the OD procedure. Assuming $\overline{G}(1\colon p, 1\colon p)$ in (2.74) is nonsingular, there exists a $J$-orthogonal $Q \in \mathbb{R}^{m\times m}$ such that*

$$\begin{array}{c} \\ p \\ q \end{array} \overset{n}{\begin{bmatrix} A_1 + \Delta A_1 \\ A_2 + \Delta A_2 \end{bmatrix}} = Q \overset{n}{\begin{bmatrix} \widehat{R}_1 \\ 0 \\ 0 \end{bmatrix}} \begin{array}{c} n \\ p-n \\ q \end{array},$$

*where*

$$\|\Delta A_1\|_F \leq \frac{\sqrt{n}}{2}(\kappa_2(A_1)\widetilde{\gamma}_{mn})\widetilde{\gamma}_{mn}\|A\|_F + O(u^3),$$
$$\|\Delta A_2\|_F \leq \widetilde{\gamma}_{mn}\|A\|_F$$

*and* $J = \mathrm{diag}(I_p, -I_q)$.

Theorem 2.5.9 shows that computing the hyperbolic QR factorization using a combination of Householder transformations and nonoverlapping hyperbolic rotations, applied in mixed form or by the OD procedure, is conditionally backward stable. In order to show that the method would not be backward stable if the hyperbolic rotations are applied directly we consider the $2 \times 2$ case with $p = 1$. The numerical experiments in Section 2.2.4 show that using hyperbolic rotation applied directly can not be mixed forward-backward stable. The proof of Lemma 2.2.8 shows that it would be possible to convert a backward stable error result to a mixed forward-backward error result of the form (2.24), and hence using a hyperbolic rotation applied directly can not be backward stable.

We now conduct further numerical experiments in order to compare the computed matrix $R$ for the various methods of applying the hyperbolic rotations. We also include tests for Algorithm 2.5.5.

To ensure the hyperbolic QR factorization exists we form $A = Q^{-1}R$, where $Q$ is $J$ orthogonal and $R$ is upper trapezoidal. Using the direct search maximisation routines of the MATLAB Matrix Computation Toolbox [27], the residual

$$\epsilon = \frac{\|A^T J A - \widehat{R}^T \widehat{R}\|_2}{\|A\|_2^2}$$

is maximised, where $A^T J A$ is computed in 100-digit arithmetic using MATLAB's Symbolic Math Toolbox.

The direct search routines can be used to naturally vary $R$, but we want to also allow $Q$ to vary while keeping it $J$-orthogonal. Higham [31] suggests

a method to generate random $J$-orthogonal matrices using the hyperbolic CS decomposition and random orthogonal matrices. Allowing an $O(n^2)$ matrix to vary, we can create varying orthogonal matrices by creating a sequence of $n-1$ Householder transformations in a similar way to how Stewart [50] creates random orthogonal matrices, except the random vectors used by Stewart are replaced with columns of the varying matrix. Using the orthogonal matrix created, we use a version of the algorithm in [31] to create a $J$-orthogonal matrix $Q$, while keeping $\|Q\|_2$ constant.

The results given in Table 2.5 for the case $m = 6$, $n = p = 5$ and $q = 1$ are surprising. It appears that the accuracy of $\widehat{R}$ is similar for all the methods of computing the hyperbolic QR factorization tested. This is despite the fact that applying hyperbolic rotations directly is unstable, and also we know of no stable methods for applying hyperbolic Householder transformations. Similar results were obtained when we tried maximising the error $\|\widehat{R} - R\|_2$. We are unable to explain this phenomenon, but would recommend using Algorithm 2.5.6 or the equivalent with the hyperbolic rotations applied by the OD procedure, since this is known to be conditionally backward stable.

Table 2.5: The maximum residuals $\beta = \|A^T JA - \widehat{R}^T \widehat{R}\|_2 / \|A\|_2^2$ in computing the hyperbolic QR factorization. The quantities $\beta_D$, $\beta_M$ and $\beta_{OD}$ are the residuals when the hyperbolic rotations are applied directly, in mixed form and by the OD procedure respectively. The quantity $\beta_H$ is the residual when using Algorithm 2.5.5. For all tests $\|\widehat{Q}\|_2 \approx \|Q\|_2$, where $\widehat{Q}$ is computed in the hyperbolic QR factorization

| $\|Q\|_2$ | $\beta_D$ | $\beta_M$ | $\beta_{OD}$ | $\beta_H$ |
|---|---|---|---|---|
| 1e2 | 6.4109e-16 | 5.5465e-16 | 5.3481e-16 | 1.3957e-16 |
| 1e4 | 8.8807e-16 | 7.0788e-16 | 5.3635e-16 | 3.4510e-16 |
| 1e6 | 8.1292e-16 | 6.8401e-16 | 6.3911e-16 | 5.4502e-16 |
| 1e8 | 7.3862e-16 | 7.5376e-16 | 7.4219e-16 | 6.8137e-16 |

### 2.5.5 Cholesky Downdating Problem

Let the positive definite matrix $A \in \mathbb{C}^{n\times n}$ have a Cholesky decomposition $A = R_1^* R_1$, where $R_1 \in \mathbb{C}^{n\times n}$ is upper triangular. The Cholesky downdating problem is to perform a rank $q$ downdate of $A$ and compute the new Cholesky decomposition of the downdated matrix. More precisely, we must compute the Cholesky decomposition of $C = R_1^* R_1 - B^* B$, where $B \in \mathbb{C}^{q\times n}$. To compute the desired Cholesky factorization efficiently we would like to take advantage of the upper triangular structure of $R_1$. We also avoid forming $C$ explicitly for numerical stability reasons.

The hyperbolic QR factorization provides one method of solving the Cholesky downdating problem. By Corollary 2.5.4, we can find a $J$-orthogonal matrix $Q$ such that

$$Q \begin{bmatrix} R_1 \\ B \end{bmatrix} = \begin{bmatrix} R \\ 0 \end{bmatrix},$$

with $J = \mathrm{diag}(I_n, -I_q)$ and $R \in \mathbb{C}^{n\times n}$ upper triangular. We observe that

$$\begin{aligned} C &= \begin{bmatrix} R_1 \\ B \end{bmatrix}^* J \begin{bmatrix} R_1 \\ B \end{bmatrix} \\ &= \begin{bmatrix} R_1 \\ B \end{bmatrix}^* Q^* J Q \begin{bmatrix} R_1 \\ B \end{bmatrix} \\ &= R^* R, \end{aligned}$$

and hence $R^* R$ is the Cholesky factorization of $C$. We note that $R_1$ need not be upper triangular and hence the hyperbolic QR factorization can be used to compute the Cholesky decomposition of $C = A^* A - B^* B$, where $A \in \mathbb{C}^{n\times n}$ and $B \in \mathbb{C}^{q\times n}$, providing $C$ is positive definite.

The following function makes use of Algorithm 2.5.6 to compute the Cholesky factor of $C$.

**Algorithm 2.5.10.** *Computes the Cholesky factor $R$ of $C = A^* A - B^* B$.*

```
1  function R =choldown(A, B)
```

2 $[Q, R] = \text{hqr}([A^T B^T]^T, n)$

The MATLAB implementation of this function is given in Appendix A.2 as the function `choldown.m`.

The built-in MATLAB function `cholupdate` can also be used to downdate a Cholesky decomposition. It computes the Cholesky factor, $R$, of the rank one downdate $C = A^*A - xx^*$, where $A \in \mathbb{C}^{n\times n}$ is upper triangular and $x \in \mathbb{C}^n$, using the method implemented in LINPACK [18] as function `zchdd`.

Instead of using hyperbolic transformations, the Cholesky factor, $R$, can be computed using the relationship

$$G_1 G_2 \ldots G_n \begin{bmatrix} A \\ 0 \end{bmatrix} = \begin{bmatrix} R \\ x^* \end{bmatrix}, \tag{2.75}$$

where $G_i$ for $i = 1\colon n$ are Givens rotations in the $(i, n+1)$ plane. The Givens rotations must be carefully chosen so that the right hand side of (2.75) has upper triangular $R$ and the desired $x$. This is achieved by solving $A^*a = x$ for $a$, setting $\alpha = \sqrt{1 - a^*a}$, and choosing the $G_i$ to satisfy

$$G_1 G_2 \ldots G_n \begin{bmatrix} a \\ \alpha \end{bmatrix} = e_{n+1},$$

where $e_{n+1}$ is the $(n+1)$st column of the identity matrix.

The algorithm used by the MATLAB function `cholupdate` is shown to be mixed forward-backward stable in [49] and extensive numerical experiments as in Table 2.5 show that this method computes $R$ as accurately as using hyperbolic transformations. Using this method to compute the Cholesky factor of the rank $q$ downdate

$$C = A^*A - B^*B,$$

where $B \in \mathbb{C}^{q\times n}$, would require $q$ applications of the algorithm and hence $qn$ Givens rotations. Since Algorithm 2.5.10 uses $n$ Householder transformations of dimension $q$, and $q$ hyperbolic rotations, it uses approximately two-thirds the

number of operations required by the algorithm used in the MATLAB function `cholupdate`.

### 2.5.6 The Indefinite Least Squares Problem

The indefinite least squares (ILS) problem is

$$\min_x (b - Ax)^T J (b - Ax),$$

where $A \in \mathbb{R}^{m \times n}$, $m \geq n$ and $b \in \mathbb{R}^m$ are given and

$$J = \begin{bmatrix} I_p & 0 \\ 0 & -I_q \end{bmatrix}, \quad p + q = m,$$

with $p \geq n$. It can be shown that there is a unique solution of the ILS problem if and only if $A^T J A$ is positive definite.

In [9] it is shown how to solve the ILS problem if there exists a hyperbolic QR factorization

$$QA = Q \begin{matrix} n \\ \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} \begin{matrix} p \\ q \end{matrix} \end{matrix} = \begin{matrix} n \\ \begin{bmatrix} R \\ 0 \end{bmatrix} \begin{matrix} n \\ m-n \end{matrix} \end{matrix},$$

where $R$ is upper triangular. By Corollary 2.5.4 there exists a matrix $Q$ if $A^T J A$ is positive definite. Then

$$Q(b - Ax) = \begin{bmatrix} d_1 - Rx \\ d_2 \end{bmatrix}, \qquad \begin{matrix} 1 \\ \begin{matrix} n \\ m-n \end{matrix} \begin{bmatrix} d_1 \\ d_2 \end{bmatrix} \end{matrix} = Qb,$$

and hence

$$\begin{aligned} (b - Ax)^T J (b - Ax) &= (b - Ax)^T Q^T J Q (b - Ax) \\ &= \begin{bmatrix} d_1 - Rx \\ d_2 \end{bmatrix}^T J \begin{bmatrix} d_1 - Rx \\ d_2 \end{bmatrix} \\ &= \|d_1 - Rx\|_2^2 + d_2^T J(n+1\colon m, n+1\colon m) d_2. \end{aligned}$$

Therefore the ILS solution can be obtained by solving $Rx = d_1$. The following function solves the ILS problem.

**Algorithm 2.5.11.** *Solves the ILS problem* $\min_x (b - Ax)^T J(b - Ax)$ *where* $A \in \mathbb{R}^{m \times n}$ *and* $J = \mathrm{diag}(I_p, -I_{m-p})$.

```
1 function x =ils(A, b, p)
2 Compute the hyperbolic QR factorization of A with respect to
    J = diag(I_p, -I_{m-p}), overwriting A(1:n,:) with R, and b with Qb.
3 R = A(1:n,:)
4 Solve Rx = b by backward substitution.
```

In [9] this method is shown, under a reasonable assumption that has been shown to be true by Grcar [23], to be forward stable. It is unclear if the method is backward stable.

A method for solving the ILS problem that uses a QR factorization and a Cholesky factorization is given by Chandrasekaran, Gu and Sayed [14]. This method is more expensive than Algorithm 2.5.11 but has been shown to be backward stable. A more efficient backward stable method that makes use of a hyperbolic QR factorization has been proposed by Xu [56]. However, this method is twice as expensive as Algorithm 2.5.11.

## 2.6 Conclusion

We have provided a detailed overview of how to compute various $J$-orthogonal and $(J_1, J_2)$-orthogonal transformations, which includes hyperbolic rotations, unified rotations, fast hyperbolic rotations and hyperbolic Householder transformations. The methods described have been implemented in MATLAB and are given in Appendix A.

Various methods of constructing and applying hyperbolic rotations have been considered, including a new stable representation (2.7) that avoids overflow. Using numerical experiments we have been able to prove that hyperbolic

rotations applied directly are unstable. We have also given new error results for applying hyperbolic rotations using the OD procedure, and have made use of these when showing how to apply a sequence of nonoverlapping hyperbolic rotations in a stable way.

We have shown how to apply fast hyperbolic rotations and unified rotations in a mixed form, and presented new error results to show that these methods are stable. New theorems have been given to show conditions for the existence of the HR factorization and the hyperbolic QR factorization, and we have shown how to compute the hyperbolic QR factorization in a stable way.

# Chapter 3

# Computing the Condition Number of Tridiagonal and Diagonal-Plus-Semiseparable Matrices in Linear Time

## 3.1 Introduction

Consider a nonsingular matrix $A \in \mathbb{R}^{n \times n}$ and the linear system $Ax = b$. The condition number of $A$,

$$\kappa(A) = \|A\|\|A^{-1}\|,$$

is often computed or estimated since it provides a measure of the sensitivity of the solution to perturbations in $A$ and $b$. The condition number depends on the choice of matrix norm. We will consider the matrix 1-norm

$$\|A\|_1 = \max_j \sum_i |a_{ij}|.$$

Various techniques exist for estimating the condition number of a general matrix in $O(n^2)$ operations, given a suitable factorization of the matrix.

Some matrices with a special structure allow the linear system to be solved in $O(n)$ operations rather than the $O(n^3)$ operations required for a general matrix. Techniques for estimating the condition number of such matrices typically reduce to $O(n)$ operations. However, the structure of the inverse may make it possible to compute the condition number exactly in $O(n)$ operations. Two types of matrices for which this can be achieved are tridiagonal and diagonal-plus-semiseparable matrices.

Tridiagonal matrices occur in many areas of numerical analysis. The inverse of a tridiagonal matrix is a semiseparable matrix, which is the sum of the strictly upper triangular part of a rank-1 matrix and the lower triangular part of another rank-1 matrix. Another link between tridiagonal and semiseparable matrices is that a symmetric matrix can be reduced to either of these forms using orthogonal transformations. This has led to new algorithms for solving the symmetric eigenvalue problem by reduction to semiseparable form [55] instead of tridiagonal form.

A diagonal-plus-semiseparable (dpss) matrix is the sum of a diagonal matrix and a semiseparable matrix. Recently several algorithms have been developed to solve $Ax = b$, where $A$ is a dpss matrix, in $O(n)$ operations [6], [13], [19], [40]. Applications of solving linear systems of this form include boundary value problems for ordinary differential equations [24], [38], [47], integral equations [36] and orthogonal rational functions [5].

The semiseparable structure of the inverse of a tridiagonal matrix is exploited by Higham to give several algorithms for computing the 1-norm of the inverse in $O(n)$ operations. However, the algorithms for a general tridiagonal matrix are prone to underflow and overflow. The algorithm for the positive definite case, which is implemented in LAPACK [3], is shown not to have such problems.

Dhillon [17] describes four algorithms based on LDU factorizations. The

first of these is less prone to underflow and overflow than Higham's algorithms but has restrictions on the tridiagonal matrix. These restrictions are eliminated using IEEE arithmetic [35] in the second algorithm, and various tests to guard against underflow and overflow problems are added to give the third and fourth algorithms. The resulting codes are complicated as they contain various tests to deal with degenerate cases.

Unlike tridiagonal matrices, we know of no existing methods for computing the condition number of a dpss matrix exactly in $O(n)$ operations.

In Section 3.2 we use the properties of the QR factorization of a tridiagonal matrix to present two new algorithms for computing the 1-norm of the inverse of a tridiagonal matrix in $O(n)$ operations. The algorithms are more elegant and simpler than Dhillon's algorithms since they do not require tests for degenerate cases. Numerical experiments show that the second algorithm is marginally slower than the quickest of Dhillon's algorithms, but is faster than his recommended algorithm. A rounding error analysis of the first algorithm is given.

In Section 3.3 we extend the techniques developed for the tridiagonal case to the dpss case, to present an algorithm that computes the 1-norm of a dpss matrix in $O(n)$ operations. The condition number of a dpss matrix can instead be estimated in $O(n)$ operations by adapting the LAPACK [3] condition number estimator to take advantage of the structure of the dpss matrix; we show, however, that our new algorithm is quicker than the possibly inaccurate estimate.

## 3.2 Tridiagonal Matrices

Let the tridiagonal matrix

$$T = \begin{bmatrix} \alpha_1 & \gamma_1 & & & \\ \beta_1 & \alpha_2 & \gamma_2 & & \\ & \beta_2 & \ddots & \ddots & \\ & & \ddots & \alpha_{n-1} & \gamma_{n-1} \\ & & & \beta_{n-1} & \alpha_n \end{bmatrix} \in \mathbb{R}^{n\times n}. \tag{3.1}$$

Since we can solve a tridiagonal system in $O(n)$ operations, we would also like to compute the condition number at the same cost. Computing $\|T\|_1$ can trivially be done in $O(n)$ operations so the problem remains to compute $\|T^{-1}\|_1$ in $O(n)$ operations.

We present two new algorithms for computing $\|T^{-1}\|_1$ in $O(n)$ operations. The algorithms use the properties of QR factorizations of tridiagonal matrices and extend some of the ideas of Bini, Gemignani and Tisseur [7] for computing the trace of the inverse of a tridiagonal matrix. The new algorithms attempt to avoid underflow and overflow without the need for tests to deal with degenerate cases as in [17].

### 3.2.1 A New Algorithm to Compute $\|T^{-1}\|_1$

The QR factorization of $T$ in (3.1) can be obtained using $n-1$ Givens rotations, $G_i$, so that

$$Q^T T = R \quad \text{and} \quad Q^T = G_{n-1} \dots G_2 G_1, \tag{3.2}$$

where $R$ is upper triangular. The Givens rotation $G_i$ is equal to the identity except for rows and columns $i$ and $i+1$, where

$$G_i([i, i+1], [i, i+1]) = \begin{bmatrix} \phi_i & \psi_i \\ -\psi_i & \phi_i \end{bmatrix}, \quad \phi_i^2 + \psi_i^2 = 1. \tag{3.3}$$

Since $T$ is tridiagonal, the upper triangular matrix $R$ is of the form

$$R = \begin{bmatrix} r_1 & s_1 & t_1 & & \\ & \ddots & \ddots & \ddots & \\ & & r_{n-2} & s_{n-2} & t_{n-2} \\ & & & r_{n-1} & s_{n-1} \\ & & & & r_n \end{bmatrix}.$$

The following theorem [21] describes the structure of $Q^T$, which allows us to compute the elements of $T^{-1}$ in $O(n)$ operations.

**Theorem 3.2.1.** *Let $T \in \mathbb{R}^{n\times n}$ be tridiagonal and unreduced and let $T = QR$ be its QR factorization computed according to* (3.2). *Define*

$$\begin{aligned} D &= \mathrm{diag}(1, -\psi_1, \psi_1\psi_2, \ldots, (-1)^{n-1}\psi_1\psi_2\cdots\psi_{n-1}), \\ u &= D^{-1}[1, \phi_1, \phi_2, \ldots, \phi_{n-1}]^T, \\ v &= D[\phi_1, \phi_2, \ldots, \phi_{n-1}, 1]^T. \end{aligned} \tag{3.4}$$

*Then*

$$Q^T = \begin{bmatrix} v_1u_1 & \psi_1 & & & 0 \\ v_2u_1 & v_2u_2 & \psi_2 & & \\ \vdots & \vdots & \ddots & \ddots & \\ \vdots & & & v_{n-1}u_{n-1} & \psi_{n-1} \\ v_nu_1 & v_nu_2 & \cdots & v_nu_{n-1} & v_nu_n \end{bmatrix}. \tag{3.5}$$

*Proof.* The proof is by induction. If $n = 2$,

$$Q_2^T = \begin{bmatrix} \phi_1 & \psi_1 \\ -\psi_1 & \phi_1 \end{bmatrix}$$

for which it is trivial to confirm that the theorem is true.

Now assume that the result is true for a tridiagonal matrix $T^{(n-1)\times(n-1)}$. Then

$$\widetilde{Q}_{n-1}^T = \begin{bmatrix} \widetilde{v}_1\widetilde{u}_1 & \psi_1 & & 0 \\ \widetilde{v}_2\widetilde{u}_1 & \ddots & \ddots & \\ \vdots & & \ddots & \psi_{n-2} \\ \widetilde{v}_{n-1}\widetilde{u}_1 & \cdots & \widetilde{v}_{n-1}\widetilde{u}_{n-2} & \widetilde{v}_{n-1}\widetilde{u}_{n-1} \end{bmatrix} \in \mathbb{R}^{(n-1)\times(n-1)},$$

where

$$\widetilde{D} = \mathrm{diag}(1, -\psi_1, \psi_1\psi_2, \ldots, (-1)^{n-2}\psi_1\psi_2\cdots\psi_{n-2}),$$
$$\widetilde{u} = \widetilde{D}^{-1}[1, \phi_1, \phi_2, \ldots, \phi_{n-2}]^T,$$
$$\widetilde{v} = \widetilde{D}[\phi_1, \phi_2, \ldots, \phi_{n-2}, 1]^T.$$

If the dimension of $T$ is increased by 1 then an extra Givens rotations is required and therefore

$$Q_n^T = G_{n-1}\begin{bmatrix} \widetilde{Q}_{n-1}^T & 0 \\ 0 & 1 \end{bmatrix} =: G_{n-1}Q_{n-1}^T.$$

Note that the first $n-2$ rows of $Q_n^T$ and $Q_{n-1}^T$ are the same, and also

$$u(1\colon n-1) = \widetilde{u}, \quad v_{n-1} = \phi_{n-1}\widetilde{v}_{n-1}, \quad v_n = -\psi_{n-1}\widetilde{v}_{n-1}, \quad \phi_{n-1} = u_n v_n.$$

If $e_k$ denotes the $k$th column of the identity matrix then

$$\begin{aligned} e_{n-1}^T Q_n^T &= \phi_{n-1}\widetilde{v}_{n-1}[\widetilde{u}^T, 0] + \psi_{n-1}e_n^T \\ &= [v_{n-1}u_1, v_{n-1}u_2, \ldots, v_{n-1}u_{n-1}, \psi_{n-1}] \end{aligned}$$

and

$$\begin{aligned} e_n^T Q_n^T &= -\psi_{n-1}\widetilde{v}_{n-1}[\widetilde{u}^T, 0] + \phi_{n-1}e_n^T \\ &= [v_n u_1, v_n u_2, \ldots, v_n u_{n-1}, v_n u_n], \end{aligned}$$

which completes the proof. □

From Theorem 3.2.1, the $(i,j)$ element of $T^{-1}$, $\eta_{ij}$, is given by

$$\eta_{ij} = e_i^T T^{-1} e_j = e_i^T R^{-1} Q^T e_j = u_j e_i^T R^{-1} v, \quad i \geq j.$$

Defining $w$ as the solution of $Rw = v$, we have

$$\eta_{ij} = u_j e_i^T w = u_j w_i, \quad i \geq j. \tag{3.6}$$

We can therefore find the elements of $T^{-1}$ in the lower triangle using (3.6).

We note that we can obtain the strictly upper triangular part of $T^{-1}$ by applying the above procedure to

$$\widetilde{T} = \begin{bmatrix} \alpha_n & \beta_{n-1} & & & \\ \gamma_{n-1} & \alpha_{n-1} & \beta_{n-2} & & \\ & \gamma_{n-2} & \ddots & \ddots & \\ & & \ddots & \alpha_2 & \beta_1 \\ & & & \gamma_1 & \alpha_1 \end{bmatrix},$$

so that

$$\widetilde{T}^{-1} = \begin{bmatrix} \eta_{nn} & & \cdots & \eta_{n1} \\ & \ddots & & \vdots \\ \vdots & & \eta_{22} & \eta_{21} \\ \eta_{1n} & \cdots & \eta_{12} & \eta_{11} \end{bmatrix}.$$

The QR factorization of $\widetilde{T}$ (which is essentially the QL factorization of $T$) is given by $\widetilde{T} = \widetilde{Q}\widetilde{R}$, where $\widetilde{Q}$ is made up of $n-1$ Givens rotations and $\widetilde{R}$ is upper triangular. By Theorem 3.2.1 the lower triangular part of $\widetilde{Q}^T$ is determined by vectors $\widetilde{u}$ and $\widetilde{v}$, where the components of $\widetilde{u}$ and $\widetilde{v}$ are defined by the $n-1$ Givens rotations. Hence the $(i,j)$ element of $\widetilde{T}^{-1}$ is given by

$$\widetilde{\eta}_{ij} = e_i^T \widetilde{T}^{-1} e_j = e_i^T \widetilde{R}^{-1} \widetilde{Q}^T e_j = \widetilde{u}_j e_i^T \widetilde{R}^{-1} \widetilde{v} = \widetilde{u}_j e_i^T \widetilde{w} = \widetilde{u}_j \widetilde{w}_i, \quad i \geq j,$$

where $\widetilde{w}$ is the solution of $\widetilde{R}\widetilde{w} = \widetilde{v}$.

The $(i,j)$ element of $T^{-1}$, $i < j$, is therefore given by

$$\eta_{ij} = \widetilde{u}_{n-j+1} \widetilde{v}_{n-i+1}.$$

We now return to the elements of $T^{-1}$ in the lower triangle. The computation of $u$ and $v$ using (3.4) may cause underflow and overflow since the diagonal entries of $D$ are products of $\psi_i$ with $|\psi_i| \leq 1$. In [7] it was noted that a way of avoiding such problems for computing the diagonal elements of $T^{-1}$ is to scale the triangular system $Rw = v$ with the diagonal matrix $D$. This gives $R'w' = v'$, where

$$R' = D^{-1}RD, \quad w' = D^{-1}w, \quad v' = D^{-1}v = [\phi_1, \ldots, \phi_{n-1}, 1]^T. \tag{3.7}$$

Using this scaling avoids products of $\psi_i$. The entries of $R'$ are given by

$$r'_i = r_i, \quad s'_i = -\psi_i s_i, \quad t'_i = \psi_i \psi_{i+1} t_i, \tag{3.8}$$

and since $|\psi_i| \leq 1$, these values cannot overflow.

Let

$$u' = [1, \phi_1, \phi_2, \ldots, \phi_{n-1}]^T. \tag{3.9}$$

As $\phi_i^2 + \psi_i^2 = 1$, the components of $u'$ are all bounded by 1 in modulus. Since $w = Dw'$ and $u = D^{-1}u'$, the diagonal elements of $T^{-1}$ are given by

$$\eta_{ii} = u_i w_i = u'_i w'_i, \quad i = 1\colon n.$$

The strictly lower triangular elements are given by

$$\eta_{ij} = u_j w_i = \frac{d_i}{d_j} u'_j w'_i, \quad i > j,$$

and hence

$$|\eta_{ij}| = |\psi_j \ldots \psi_{i-1} u'_j w'_i|, \quad i > j,$$

for which we clearly have possible underflow problems if $\eta_{ij}$ is computed in this way.

Fortunately, this problem can be overcome by considering how to compute the 1-norm of the $j$th column of the strictly lower triangular part of $T^{-1}$, which is given by

$$\sigma_j = |u'_j|(|w'_{j+1}\psi_j| + |w'_{j+2}\psi_j\psi_{j+1}| + \cdots + |w'_n \psi_j \ldots \psi_{n-1}|), \quad j = 1\colon n-1.$$

If $\sigma_j$ is computed in this way $O(n^2)$ operations would be required to compute $\sigma_j$ for $j = 1\colon n-1$. The underflow problem can be overcome and the operation count reduced by an order of magnitude using nested multiplication. We can rewrite $\sigma_j$ as

$$\sigma_j = |u'_j|(((\cdots((|\psi_{n-1}w'_n| + |w'_{n-1}|)|\psi_{n-2}| + |w'_{n-2}|)\cdots)|\psi_{j+1}| + |w'_{j+1}|)|\psi_j|), \quad j = 1\colon n-1.$$

Therefore $\sigma_j$ can be computed using

```
1  σ'_{n-1} = |ψ_{n-1} w'_n|
2  for j = n − 1: −1: 2
3        σ'_{j-1} = (σ'_j + |w'_j|)|ψ_{j-1}|
4  end                                                    (3.10)
5  for j = 1: n − 1
6        σ_j = |u'_j| σ'_j
7  end
```

This method of computing $\sigma_j$, avoids forming explicit products of $\psi_i$ and allows $\sigma_j$, $j = 1{:}\,n-1$, to be computed in $O(n)$ operations. For underflow to occur $\sigma_j/|u'_j|$ must underflow, and since $\sigma_j/|u'_j| \geq \sigma_j$, this can only occur if $\sigma_j$, the 1-norm of the $j$th column of the strictly lower triangular part of $T^{-1}$, underflows. We note that this is very unlikely and that underflow does not cause the algorithm to break down.

When considering $\widetilde{T}$, we can use a similar approach that also adds the absolute values of the diagonal elements of $T^{-1}$. This enables us to compute the 1-norm of the $j$th column of the upper triangular part of $T^{-1}$, denoted by $\widetilde{\sigma}_j$, in $O(n)$ operations without the risk of underflow. Defining $\sigma_n = 0$ the 1-norm of $T^{-1}$ is then given by

$$\|T^{-1}\|_1 = \max_{j=1:n}(\sigma_j + \widetilde{\sigma}_j).$$

The pseudocode in the following algorithm summarises the method described above. The function $(\phi, \psi, r) = Givens(a, b)$ computes $r = \sqrt{a^2 + b^2}$, $\phi = a/r$ and $\psi = b/r$, and guards against overflow.

**Algorithm 3.2.2.** *Computes* $\tau = \|T^{-1}\|_1$, *where* $T$ *is given by* (3.1).

```
1  for k = 1: 2
2        a = α_1,  g = γ_1,  u'_1 = 1
3        for i = 1: n − 1
4              (φ_i, ψ_i, r'_i) = Givens(a, β_i)
```

5 if $r'_i = 0$, $\tau = \infty$, return, end
6 $s'_i = -\psi_i(\phi_i g + \psi_i \alpha_{i+1})$
7 $a = -\psi_i g + \phi_i \alpha_{i+1},\ u'_{i+1} = \phi_i,\ v'_i = \phi_i$
8 if $i < n-1,\ t_i = \psi_i^2 \gamma_{i+1},\ g = \phi_i \gamma_{i+1}$, end
9 if $i > 1,\ t'_{i-1} = \psi_i t'_{i-1}$, end
10 end
11 $r'_n = a,\ v'_n = 1$
12 if $r'_n = 0$, $\tau = \infty$, return, end
13 $w'_n = v'_n / r'_n$
14 $w'_{n-1} = (v'_{n-1} - w'_n s'_{n-1}) / r'_{n-1}$
15 for $i = n-2\colon -1\colon 1$
16 $w'_i = (v'_i - w'_{i+1} s'_i - w'_{i+2} t'_i) / r'_i$
17 end
18 if $k = 1$
19 Compute $\sigma_j$ using the code given in (3.10)
20 $\alpha = \alpha(n\colon -1\colon 1),\ \delta = \beta,\ \beta = \gamma(n-1\colon -1\colon 1), \gamma = \delta(n-1\colon -1\colon 1)$
21 else
22 $\widetilde{\sigma}_n = |w'_n|$
23 for $i = n-1\colon -1\colon 1$
24 $\widetilde{\sigma}_i = \widetilde{\sigma}_{i+1} |\psi_i| + |w'_i|$
25 end
26 for $i = 1\colon n-1$, $\widetilde{\sigma}_i = \widetilde{\sigma}_i |u'_i|$, end
27 end
28 end
29 $\tau = \max_{i=1:n}(\sigma_i + \widetilde{\sigma}_{n-i+1})$

**Cost**: Assuming the function *Givens* requires 6 operations, the total cost is $51n$ operations.

Lines 2–12 compute the vectors $r'$, $s'$, $t'$ in (3.8), $u'$ in (3.9) and $v'$ in (3.7) and lines 13–17 solve the linear system $R'w' = v'$. The 1-norms of the columns of the strictly lower and the upper triangular parts of $T^{-1}$ are computed in lines 19 and 22–26 respectively. The only divisions in the algorithm are by $r_i$. Hence the test on line 5 prevents division by zero, which is possible if $T$ is singular.

Algorithm 3.2.2 does not have to deal with the reduced case (defined in

the next section) separately as in [28], or have numerous tests to deal with the reduced case as in [17]. It can also be easily adapted to compute $\|T^{-1}\|_1$ for $T \in \mathbb{C}^{n\times n}$ by using complex Givens rotations and the complex conjugate of $v'$ in (3.7).

We note that there are certain similarities between Algorithm 3.2.2 and those in [17]. All the algorithms use factorizations: triangular factorizations in [17] and orthogonal factorizations in Algorithm 3.2.2. Also, Algorithm 3.2.2 and the algorithms in [17] both compute two factorizations, with the first proceeding from the top of $T$ to the bottom and the second from the bottom of $T$ to the top. In Algorithm 3.2.2 this takes the form of a QR factorization followed by a QL factorization, whereas in [17] the factorizations $T = L_+D_+U_+$ and $T = L_-D_-U_-$ are computed, where $L_+$ and $L_-$ are lower bidiagonal, $U_+$ and $U_-$ upper bidiagonal, and $D_+$ and $D_-$ diagonal.

### 3.2.2 Reducing the Operation Count

The matrix $T$ in (3.1) is said to be unreduced if $\beta_i\gamma_i \neq 0$ for $i = 1\colon n-1$. If $T$ is unreduced then it can be scaled using the diagonal matrix

$$D = \operatorname{diag}(d_i), \quad d_1 = 1, \quad d_i = \frac{\beta_1 \dots \beta_{i-1}}{\gamma_1 \dots \gamma_{i-1}}, \quad i > 1,$$

so that $\widetilde{T} = TD$ is symmetric. In this section we will use this property to obtain an algorithm for computing $\|T^{-1}\|_1$ that requires approximately half the number of operations of Algorithm 3.2.2.

We first note that it is also possible to use a similar scaling so that $\check{T} = \check{D}T$ is symmetric, where $\check{D}$ is diagonal and is defined by products of $\gamma_i/\beta_i$. A naive method of computing $\|T^{-1}\|_1$ would be to use this scaling to form $\check{T}$, compute the absolute column sums of $\check{T}^{-1}$ using the ideas of Section 3.2.2 and making use of its symmetry, and then recover the absolute column sums of $T^{-1}$ using $T^{-1} = \check{T}^{-1}\check{D}$. Explicitly forming $\check{T}$ in this approach can lead to underflow and

overflow due to the computation of the elements of $\check{D}$. We will use the scaling $\widetilde{T} = TD$ without forming $\widetilde{T}$ and use the structure of $T^{-1}$ to deal with reduced $T$.

The QR factorization of $\widetilde{T}$ has the form

$$Q^T\widetilde{T} = RD,$$

where $Q$ and $R$ satisfy $Q^TT = R$ and can be computed using (3.4). The matrix $Q$ is defined by (3.5). The $(i,j)$ element of $\widetilde{T}^{-1}$, $\widetilde{\eta}_{ij}$, is therefore given by

$$\widetilde{\eta}_{ij} = d_i^{-1}u_jw_i,$$

where $w$ is the solution of the triangular system $Rw = v$.

In Section 3.2.1 we showed how to scale the system $Rw = v$ to avoid underflow and overflow. Using the same procedure here we find that

$$|\widetilde{\eta}_{ij}| = |d_i^{-1}\psi_j \ldots \psi_{i-1}u'_jw'_i|, \quad i \geq j,$$

where $u'$ is defined by (3.9) and $w'$ is defined by (3.7).

Since $T^{-1} = D\widetilde{T}^{-1}$ the $(i,j)$ element of $T^{-1}$, $\eta_{ij}$, satisfies

$$|\eta_{ij}| = |\psi_j \ldots \psi_{i-1}u'_jw'_i|, \quad i \geq j,$$

which is precisely what we found in Section 3.2.1. Therefore the 1-norm of the $j$th column of the strictly lower triangle of $T^{-1}$, $\sigma_j$, can be calculated as described in Section 3.2.1.

Using the symmetry of $\widetilde{T}^{-1}$ the upper triangular elements of $\widetilde{T}^{-1}$ satisfy

$$|\widetilde{\eta}_{ij}| = |d_j^{-1}\psi_i \ldots \psi_{j-1}u'_iw'_j|, \quad i \leq j,$$

and therefore using $T^{-1} = D\widetilde{T}^{-1}$

$$\begin{aligned} |\eta_{ij}| &= \left| \frac{d_i}{d_j}\psi_i \ldots \psi_{j-1}u'_iw'_j \right|, \quad i \leq j, \\ &= \left| \frac{\gamma_i \ldots \gamma_{j-1}}{\beta_i \cdots \beta_{j-1}}\psi_i \ldots \psi_{j-1}u'_iw'_j \right|. \end{aligned} \tag{3.11}$$

Using nested multiplication, similar to that described for computing $\sigma_j$, we can compute the 1-norm of the $j$th column of the upper triangle of $T^{-1}$, $\widetilde{\sigma}_j$, using

$$
\begin{array}{ll}
1 & \widetilde{\sigma}_1 = |u_1'| \\
2 & \text{for } j = 1\colon n-1 \\
3 & \qquad \widetilde{\sigma}_{j+1} = \widetilde{\sigma}_j |\gamma_j \psi_j / \beta_j| + |u_{j+1}'| \\
4 & \text{end} \\
5 & \text{for } j = 1\colon n \\
6 & \qquad \widetilde{\sigma}_j = |w_j'| \widetilde{\sigma}_j \\
7 & \text{end}
\end{array}
\tag{3.12}
$$

Computing $\widetilde{\sigma}_j$ in this way allows us to make use of the lower triangular part of $T^{-1}$ to find the upper triangular part and therefore approximately halves the number of operations required compared with the method of the previous section. We note that we encounter division by zero when computing $\widetilde{\sigma}_j$ if $\beta_j = 0$ and hence the procedure described above is only valid for tridiagonal matrices with all $\beta_j \neq 0$. However, it is possible to remove this restriction by considering the structure of $T$ when $b_k = 0$ for some $k$.

If the tridiagonal matrix $T$ has $\beta_k = 0$ then we can write it as

$$
T = \begin{bmatrix} T_1 & C \\ 0 & T_2 \end{bmatrix}, \quad T_1 \in \mathbb{R}^{k \times k}, \quad T_2 \in \mathbb{R}^{(n-k)\times(n-k)}, \quad C \in \mathbb{R}^{k \times (n-k)},
$$

where $C = \gamma_{k-1} e_k e_1^T$ and $e_j$ denotes the $j$th unit vector. The inverse of the tridiagonal matrix is then given by

$$
T^{-1} = \begin{bmatrix} T_1^{-1} & -T_1^{-1} C T_2^{-1} \\ 0 & T_2^{-1} \end{bmatrix},
$$

and

$$
T_1^{-1} C T_2^{-1} = \gamma_{k-1} (T_1^{-1} e_k)(T_2^{-1} e_1)^T. \tag{3.13}
$$

We note that the $k$th absolute column sum of $xy^T$, where $x, y \in \mathbb{R}^n$, is $\|x\|_1 |y_k|$ and therefore we can find the absolute column sums of (3.13) using the last absolute column sum of $T_1^{-1}$ and the first row of $T_2^{-1}$.

Suppose we are using the procedure described in (3.12) to compute $\widetilde{\sigma}_j$. Then if we encounter a $\beta_j$ that is zero at line 3, the last absolute column sum of $T_1^{-1}$ is given by $w_j'\widetilde{\sigma}_j$. The absolute values of the entries of the upper triangular part of $T_2^{-1}$ are given by (3.11). Since we know the last absolute column sum of $T_1^{-1}$ and the absolute values of the first row of $T_2^{-1}$, we can combine these and the absolute values of the upper triangular part of $T_2^{-1}$ to give the 1-norm of the $j$th column of the upper triangular part of $T^{-1}$ for $j = k+1\colon n$. This is achieved by setting

$$\widetilde{\sigma}_{j+1} = |\widetilde{\sigma}_j w_j \gamma_j u_{j+1}'| + |u_{j+1}'|,$$

instead of line 3. The $|u_{j+1}'|$ term corresponds to the 1-norm of the upper triangular part of $T_2^{-1}$ and the other term corresponds to the 1-norm of the columns of (3.13). Clearly these ideas apply to $T_1^{-1}$ and $T_2^{-1}$ and therefore the procedure will work if there are several $\beta_k$ equal to zero. We summarise this method of computing the absolute column sums of the upper triangular part of $T^{-1}$ as follows:

```
1  σ̃_1 = |u_1'|
2  for j = 1:n-1
3       if β_j = 0
4            σ̃_{j+1} = |σ̃_j w_j' γ_j u_{j+1}'| + |u_{j+1}'|
5       else
6            σ̃_{j+1} = σ̃_j |γ_j ψ_j/β_j| + |u_{j+1}'|
7       end
8  end
9  for j = 1:n
10      σ̃_j = |w_j'| σ̃_j
11 end
```
(3.14)

Defining $\sigma_n = 0$, we have $\|T^{-1}\|_1 = \max_{j=1:n}(\sigma_j + \widetilde{\sigma}_j)$. The following algorithm computes $\|T^{-1}\|_1$ using the method described in this section.

**Algorithm 3.2.3.** *Computes* $\tau = \|T^{-1}\|_1$, *where* $T$ *is given by* (3.1).

1 Use lines 2–12 and 19 in Algorithm 3.2.2 to compute $\sigma_i$
2 Compute $\widetilde{\sigma}_i$ using the code in (3.14)
3 $\tau = \max_{i=1:n}(\sigma_i + \widetilde{\sigma}_i)$

**Cost**: Assuming the function *Givens* requires 6 operations, the total cost is $31n$ operations.

Algorithms 3.2.2 and 3.2.3 have been implemented in the style of an LAPACK routine, and are given in Appendix B. The codes are written in Fortran 77 with careful consideration so as to maximise the speed of the algorithms. The equivalent codes for the complex case are also given.

### 3.2.3 Rounding Error Analysis

A rounding error analysis of Algorithm 3.2.2 is given. We have been unable to give a rounding error analysis of Algorithm 3.2.3 since when considering the upper triangular part of $T^{-1}$ we obtain error results which are bounded by the norm of $\widetilde{T} = TD$, which cannot be bounded a priori. However, extensive numerical experiments suggest that Algorithm 3.2.3 behaves in a stable way.

Given a tridiagonal matrix $T \in \mathbb{R}^{n\times n}$, the first step of the algorithm is to compute the QR factorization of $T$. From [30, Thm. 19.10] we have that there exists an orthogonal $Q$ such that the computed upper triangular matrix $\widehat{R}$ satisfies

$$T + \Delta T = Q\widehat{R}, \quad \|\Delta T\|_F \le \widetilde{\gamma}_n \|T\|_F. \tag{3.15}$$

We make the simplifying assumption that the defining variables of the $n-1$ Givens rotations, $\phi_1, \ldots, \phi_{n-1}$ and $\psi_1, \ldots, \psi_{n-1}$, are computed exactly. Using $D = \mathrm{diag}(d_1, \ldots, d_n)$, where $D$ is given by (3.4), the components of the computed scaled upper triangular matrix $\widehat{R}'$ (3.8) satisfy

$$\begin{aligned} \widehat{r}'_i &= \widehat{r}_i, \\ \widehat{s}'_i &= \frac{d_{i+1}}{d_i}\widehat{s}_i(1+\Delta s_i), \quad |\Delta s_i| \le \gamma_1, \end{aligned}$$

$$\widehat{t}_i' = \frac{d_{i+2}}{d_i}\widehat{t}_i(1+\Delta t_i), \quad |\Delta t_i| \le \gamma_2.$$

We now consider solving the triangular system $\widehat{R}'w' = \widehat{v}'$ and the errors involved in solving the system using backward substitution. Since we have made the assumption that $\phi_1, \ldots, \phi_{n-1}$ are computed exactly, the vectors $v' = D^{-1}v = [\phi_1, \ldots, \phi_{n-1}, 1]^T$ and $u' = Du = [1, \phi_1, \ldots, \phi_{n-1}]^T$ must also be computed exactly. The last component of the computed solution to the triangular system therefore satisfies

$$\widehat{r}_n\widehat{w}_n'(1+\theta_1) = \frac{v_n}{d_n},$$

which simplifies to

$$d_n\widehat{r}_n(1+\widetilde{\Delta r}_n)\widehat{w}_n' = v_n, \quad |\widetilde{\Delta r}_n| \le \gamma_1.$$

The computed component $\widehat{w}_{n-1}'$ satisfies

$$\widehat{r}_{n-1}\widehat{w}_{n-1}'(1+\theta_1') = \frac{v_{n-1}}{d_{n-1}} - \widehat{w}_n'\frac{d_n}{d_{n-1}}\widehat{s}_{n-1}(1+\Delta s_{n-1})(1+\theta_3),$$

which simplifies to

$$d_{n-1}\widehat{r}_{n-1}(1+\widetilde{\Delta r}_{n-1})\widehat{w}_{n-1}' = v_n - \widehat{w}_n'd_n\widehat{s}_{n-1}(1+\widetilde{\Delta s}_{n-1}),$$

where $|\widetilde{\Delta r}_{n-1}| \le \gamma_1$ and $|\widetilde{\Delta s}_{n-1}| \le \gamma_4$.

Similarly for $i = 1\colon n-2$, $\widehat{w}_i'$ satisfies

$$d_i\widehat{r}_i(1+\widetilde{\Delta r}_i)\widehat{w}_i' = v_i - \widehat{w}_{i+1}'d_{i+1}\widehat{s}_i(1+\widetilde{\Delta s}_i) - \widehat{w}_{i+2}'d_{i+2}\widehat{t}_i(1+\widetilde{\Delta t}_i),$$

where

$$|\widetilde{\Delta r}_i| \le \gamma_1, \quad |\widetilde{\Delta s}_i| \le \gamma_4, \quad |\widetilde{\Delta t}_i| \le \gamma_5.$$

Combining the above results for all the components of $\widehat{w}'$ and using Lemma 1.2.1 gives

$$(\widehat{R} + \widetilde{\Delta R})D\widehat{w}' = v, \tag{3.16}$$

where

$$\|\widetilde{\Delta R}\|_F \leq \gamma_5 \|\widehat{R}\|_F \leq \gamma_5 \|T\|_F + O(u^2).$$

By considering the errors in computing $\sigma_j$ using nested multiplication as described in Section 3.2.1, the computed 1-norm of the $j$th column of the lower triangular part of $T^{-1}$ is found to satisfy

$$\widehat{\sigma}_j = \left( \sum_{i=j}^{n} |\widehat{\eta}_{ij}| \right) (1 + \gamma_{2n-2}), \tag{3.17}$$

where

$$\begin{aligned}
\widehat{\eta}_{ij} &= (-1)^{i+j} \psi_j \dots \psi_{i-1} \widehat{u}'_j \widehat{w}'_i \\
&= \frac{d_i}{d_j} \widehat{u}'_j \widehat{w}'_i \\
&= d_i u_j e_i^T \widehat{w}' \\
&= d_i u_j e_i^T D^{-1} (\widehat{R} + \widetilde{\Delta R})^{-1} v \quad \text{using (3.16)} \\
&= e_i^T (\widehat{R} + \widetilde{\Delta R})^{-1} Q^T e_j.
\end{aligned}$$

Rearranging (3.15) gives $\widehat{R} = Q^T(T + \Delta T)$ and hence

$$\begin{aligned}
\widehat{\eta}_{ij} &= e_i^T (Q^T (T + \Delta T) + \widetilde{\Delta R})^{-1} Q^T e_j \\
&= e_i^T (Q^T (T + \Delta T + Q\widetilde{\Delta R}))^{-1} Q^T e_j \\
&= e_i^T (T + \Delta T + Q\widetilde{\Delta R})^{-1} e_j \\
&= e_i^T (T + \widetilde{\Delta T})^{-1} e_j,
\end{aligned}$$

where

$$\|\widetilde{\Delta T}\|_F = \|\Delta T + Q\widetilde{\Delta R}\|_F \leq \widetilde{\gamma}_n \|T\|_F.$$

Using Lemma 1.2.2 we have

$$\|\widetilde{\Delta T}\|_1 \leq n\widetilde{\gamma}_n \|T\|_1.$$

Using the result [48]

$$\frac{\|A^{-1} - (A+E)^{-1}\|_1}{\|A^{-1}\|_1} \leq \frac{e}{1-e}, \quad e = \kappa_1(A) \frac{\|E\|_1}{\|A\|_1} < 1,$$

and defining $(\widehat{T}^{-1})_{ij} = \widehat{\eta}_{ij}$ we have

$$\frac{\|\widehat{T}^{-1} - T^{-1}\|_1}{\|T^{-1}\|_1} = \frac{\|(T + \widetilde{\Delta T})^{-1} - T^{-1}\|_1}{\|T^{-1}\|_1}$$
$$\leq \frac{e}{1-e}, \quad e = \kappa_1(T)\frac{\|\widetilde{\Delta T}\|_1}{\|T\|_1} < 1.$$

Therefore

$$\|\widehat{T}^{-1} - T^{-1}\|_1 \leq \kappa_1(T)\frac{\|\widetilde{\Delta T}\|_1\|T^{-1}\|_1}{\|T\|_1}$$
$$\leq n\widetilde{\gamma}_n\kappa_1(T)\|T^{-1}\|_1. \tag{3.18}$$

Using (3.17) and (3.18)

$$|\widehat{\sigma}_j - \sigma_j| \approx \left|\sum_{i=j}^{n}(|\widehat{\eta}_{ij}| - |\eta_{ij}|)\right|$$
$$\leq \sum_{i=j}^{n}|\widehat{\eta}_{ij} - \eta_{ij}|$$
$$\leq n\widetilde{\gamma}_n\kappa_1(T)\|T^{-1}\|_1.$$

A similar result holds for the 1-norm of the columns of the upper triangular part of $T^{-1}$. Hence if we denote by $\widehat{\tau}$, our approximation to $\|T^{-1}\|_1$, computed in floating point arithmetic, we have

$$\frac{|\widehat{\tau} - \|T^{-1}\|_1|}{\|T^{-1}\|_1} \leq 2n\widetilde{\gamma}_n\kappa_1(T). \tag{3.19}$$

This error result is the best we can expect, since it can be shown that the condition number of computing the condition number is the condition number [25].

### 3.2.4 Numerical Experiments

We compare the accuracy of our new algorithms against Dhillon's recommended algorithm *nrminv_final2*, Algorithm 4.2 from [28] and MATLAB's `cond`, which computes the condition number of a matrix in $O(n^3)$ operations. The speed of the new algorithms is tested with Dhillon's *nrminv_final2* but also the quicker algorithm *nrminv_final1*. The test matrices are described in Table 3.1.

Table 3.1: Test matrices.

| Matrix Type | Description |
|---|---|
| 1 | Nonsymmetric random tridiagonal, elements uniformly distributed in $[-1, 1]$. |
| 2 | `gallery('randsvd',100,1e15,2,1,1)` in MATLAB, which creates a random tridiagonal matrix with all singular values close to 1 except for one that is of order $10^{-15}$, so that the 2-norm condition number is $10^{15}$. |
| 3 | `gallery('randsvd',100,1e15,3,1,1)` in MATLAB, which creates a random tridiagonal matrix with geometrically distributed singular values and 2-norm condition number $10^{15}$. |
| 4 | Tridiagonal with $\alpha_i = 10^8$, $\beta_i = \gamma_i = 1$. |
| 5 | Tridiagonal with $\alpha_i = 10^{-8}$, $\beta_i = \gamma_i = 1$. |
| 6 | `gallery('lesp',100)` in MATLAB. |
| 7 | `gallery('dorr',100,1e-4)` in MATLAB. |
| 8 | Nonsymmetric tridiagonal, elements uniformly distributed in $[-1, 1]$ except $\beta_{50}$ given by $1 \times 10^{-50}$ multiplied by a random number in $[-1, 1]$. |
| 9 | Nonsymmetric tridiagonal, elements of $\alpha$ and $\gamma$ uniformly distributed in $[-1, 1]$ and $\beta_i$ given by $1 \times 10^{-50}$ multiplied by a random number in $[-1, 1]$, for $i = 1\!:\!n$. |
| 10 | Symmetric tridiagonal with $\alpha_i = 0$, $\beta_i = \gamma_i = 1$. |

The results, given in Table 3.2, show that the new algorithms give the results up to four decimal place of accuracy for all but test matrix 7. Also, the new algorithms do not suffer from the underflow and overflow problems of Higham's algorithm, which breaks down for test matrices 2, 4, 6, and 9 due to overflow. All the algorithms correctly detect the singular test matrix 10. The difference in results for test matrix 7 are due to rounding errors, and in fact all the methods tested are inaccurate since the actual condition number is $8.885 \times 10^{24}$. As noted in Section 3.2.3, the best error bound we can expect to obtain for computing the condition number is of the form (3.19). In this case the bound (3.19) is approximately $10^{34}$ which suggests that the computed condition number may be inaccurate.

Table 3.2: Computation of $\kappa_1(T)$ on test matrices. Test matrices 1 to 9 are of order 100 and test matrix 10 is of order 99.

| Matrix Type | Higham's algorithm | *nrminv_final2* | Algorithm 3.2.2 | Algorithm 3.2.3 | MATLAB's `cond` |
|---|---|---|---|---|---|
| 1 | 2.9946e+03 | 2.9946e+03 | 2.9946e+03 | 2.9946e+03 | 2.9946e+03 |
| 2 | NaN | 1.4979e+15 | 1.4979e+15 | 1.4979e+15 | 1.4979e+15 |
| 3 | 4.5336e+15 | 4.5336e+15 | 4.5336e+15 | 4.5336e+15 | 4.5336e+15 |
| 4 | NaN | 1.0000 | 1.0000 | 1.0000 | 1.0000 |
| 5 | 100.0000 | 100.0000 | 100.0000 | 100.0000 | 100.000 |
| 6 | NaN | 67.1164 | 67.1164 | 67.1164 | 67.1164 |
| 7 | 2.3712e+19 | 2.7617e+19 | 1.9403e+19 | 2.1895e+19 | 3.5211e+19 |
| 8 | 1.0568e+04 | 1.0568e+04 | 1.0568e+04 | 1.0568e+04 | 1.0568e+04 |
| 9 | NaN | 6.1603e+10 | 6.1603e+10 | 6.1603e+10 | 6.1603e+10 |
| 10 | $\infty$ | $\infty$ | $\infty$ | $\infty$ | $\infty$ |

Before we consider the times taken by the various algorithms we first consider the cost. For Algorithms 3.2.2 and 3.2.3, this depends on how the function *Givens* is implemented. In our experiments we use the LAPACK [3] routine `DLARTG` in place of *Givens*, which requires at most 10 operations but typically 6. The maximum number of operations required by the algorithms for computing $\|T^{-1}\|_1$ are given in Table 3.3.

Table 3.3: The maximum number of operations required to compute the 1-norm of the inverse of an $n \times n$ tridiagonal matrix for Dhillon's algorithms and the algorithms presented here.

| Algorithm | *nrminv_final1* | *nrminv_final2* | Algorithm 3.2.2 | Algorithm 3.2.3 |
|---|---|---|---|---|
| Total | $22n$ | $27n$ | $59n$ | $35n$ |

Table 3.4 shows the times taken for the new algorithms and Dhillon's algorithms to compute $\kappa_1(T)$ for nonsymmetric random tridiagonal matrices. The algorithms were implemented in Fortran 77 and compiled using the NAGWare Fortran 95 compiler with the normal optimisation level (`-O2`) and IEEE

arithmetic. The machine used was a 2010MHz AMD Athlon machine running Linux.

The results show that *nrminv_final1* is the quickest as expected. However, Algorithm 3.2.3 runs much more quickly than the operation count would suggest, which could be due to the fewer if-statements required. As a result, Algorithm 3.2.3 is only marginally slower than *nrminv_final1*. However Algorithm 3.2.3 is quicker than *nrminv_final2*, which is the recommended algorithm in [17] due to less element growth in its computation. Interestingly, Algorithm 3.2.3 is as quick as *nrminv_final1* if both are compiled with no optimisation (`-O0`).

Table 3.4: Time taken in seconds to compute the 1-norm of the inverse of a random $n \times n$ tridiagonal matrix, for various $n$.

| Dimension | $10^6$ | $2 \times 10^6$ | $4 \times 10^6$ | $6 \times 10^6$ | $8 \times 10^6$ |
|---|---|---|---|---|---|
| *nrminv_final1* | 0.31 | 0.60 | 1.22 | 1.78 | 2.54 |
| *nrminv_final2* | 0.37 | 0.73 | 1.45 | 2.17 | 3.08 |
| Algorithm 3.2.2 | 0.60 | 1.20 | 2.37 | 3.54 | 4.87 |
| Algorithm 3.2.3 | 0.33 | 0.65 | 1.26 | 1.93 | 2.72 |

## 3.3 Diagonal-Plus-Semiseparable Matrices

A semiseparable matrix, $S$, is the sum of the strictly upper triangular part of a rank-1 matrix and the lower triangular part of another rank-1 matrix,

$$S = \text{tril}(qp^T, 0) + \text{triu}(xy^T, 1).$$

Here, $\text{tril}(A, i)$ denotes a matrix $A$ with the entries above the $i$th diagonal set to zero, where $i = 0$ is the leading diagonal and $i > 0$, $i < 0$, is above and below the leading diagonal respectively. Similarly $\text{triu}(A, i)$ denotes $A$ with the entries below the $i$th diagonal set to zero.

A diagonal-plus-semiseparable (dpss) matrix, $A$, is the sum of a diagonal matrix and a semiseparable matrix:

$$A = \mathrm{diag}(z) + S = \begin{bmatrix} p_1q_1 + z_1 & x_1y_2 & \cdots & x_1y_n \\ p_1q_2 & \ddots & \ddots & \vdots \\ \vdots & \ddots & & x_{n-1}y_n \\ p_1q_n & \cdots & p_{n-1}q_n & p_nq_n + z_n \end{bmatrix}. \tag{3.20}$$

We assume that $A$ is nonsingular.

Assuming that the defining vectors $p$, $q$, $x$, $y$ and $z$ are given, we show how to compute the condition number of $A$ in $O(n)$ operations. This is achieved in a similar way to the tridiagonal case in Section 3.2.1. We will again make use of the special structure of $Q$ and $R$ in the QR factorization of $A$, solve a scaled linear system and form absolute column sums using methods that avoid underflow and overflow.

We first note that it is possible to compute $\|A\|_1$ in $O(n)$ operations since the $i$th absolute column sum can be computed using

$$|y_i|(|x_1| + |x_2| + \cdots + |x_{i-1}|) + |p_i|(|q_{i+1}| + |q_{i+2}| + \cdots + |q_n|) + |p_iq_i + z_i|.$$

By accumulating the sums $|x_1| + |x_2| + \cdots + |x_{i-1}|$ and $|q_{i+1}| + |q_{i+2}| + \cdots + |q_n|$ we can compute the absolute column sums and hence $\|A\|_1$ in $O(n)$ operations. The problem remains to compute $\|A^{-1}\|_1$.

### 3.3.1 Computing the 1-Norm of the Inverse of a DPSS Matrix

The QR factorization of a diagonal-plus-semiseparable matrix can be considered in two stages. First, it is shown in [6] that by applying Givens rotations and scaling we can find an orthogonal matrix $Q_1 \in \mathbb{R}^{n\times n}$ so that $Q_1^TA = H$, where $H$ is upper Hessenberg. We require the condition $q_n \neq 0$, however we later show that if $q_n = 0$ we need only consider a principal submatrix of $A$. The

special structure of $H$ and $Q_1$ is described in the following proposition which summarises the result from [6].

**Proposition 3.3.1.** *The upper Hessenberg matrix $H = Q_1^T A$ has an upper triangular part that is equal to the upper triangular part of the rank-2 matrix $ab^T + fg^T \in \mathbb{R}^{n\times n}$ and subdiagonal $h \in \mathbb{R}^{n-1}$ , where*

$$a = \begin{bmatrix} 1 \\ q_1 \\ q_2 \\ \vdots \\ q_{n-1} \end{bmatrix}, \quad b = \begin{bmatrix} p_1\mu_1 + z_1q_1 \\ (q_1x_1)y_2 + p_2\mu_2 + z_2q_2 \\ (\sum_{i=1}^{2} q_ix_i)y_3 + p_3\mu_3 + z_3q_3 \\ \vdots \\ (\sum_{i=1}^{n-1} q_ix_i)y_n + p_n\mu_n + z_nq_n \end{bmatrix}, \tag{3.21}$$

$$f = \begin{bmatrix} 0 \\ -\mu_1x_1 \\ -(\sum_{i=1}^{1} q_ix_i)q_2 - \mu_2x_2 \\ \vdots \\ -(\sum_{i=1}^{n-2} q_ix_i)q_{n-1} - \mu_{n-1}x_{n-1} \end{bmatrix}, \quad g = y, \quad h = \begin{bmatrix} -z_1\mu_2 \\ -z_2\mu_3 \\ \vdots \\ -z_{n-1}\mu_n \end{bmatrix}, \tag{3.22}$$

*with $\mu_i = q_i^2 + \cdots + q_n^2$. The orthogonal matrix $Q_1^T$ is upper Hessenberg with an upper triangular part that is equal to the upper triangular part of the rank-1 matrix $rs^T \in \mathbb{R}^{n\times n}$ and subdiagonal $t \in \mathbb{R}^{n-1}$ , where*

$$r = \begin{bmatrix} 1 \\ q_1 \\ q_2 \\ \vdots \\ q_{n-1} \end{bmatrix}, \quad s = q, \quad t = \begin{bmatrix} -\mu_2 \\ -\mu_3 \\ \vdots \\ -\mu_n \end{bmatrix}. \tag{3.23}$$

We note that the defining vectors of $H$ and $Q_1^T$ can all be computed in $O(n)$ operations.

The second stage is to reduce $H$ to upper triangular form, which can be achieved by $n-1$ Givens rotations, $G_i$, defined by (3.3).

The first Givens rotation $G_1$ is chosen to zero the first element of the subdiagonal of $H$, and is therefore defined by $\phi_1 = (a_1b_1 + f_1g_1)/\tau_1$ and $\psi_1 = h_1/\tau_1$

with $\tau_1 = \sqrt{(a_1 b_1 + f_1 g_1)^2 + h_1^2}$. Since $G_1$ zeros $h_1$ and the upper triangular part of $H$ is the upper triangular part of $ab^T + fg^T$, we apply $G_1$ to $a$ and $f$ to give $a^{(1)} = G_1 a$ and $f^{(1)} = G_1 f$. Except for the first diagonal element, the upper triangular part of $G_1 H$ is the upper triangular part of the rank-2 matrix $a^{(1)} b^T + f^{(1)} g^T$. The first diagonal element is given by $\tau_1$.

This can be repeated to zero the $i$th subdiagonal element of $G_{i-1} \ldots G_1 H$ by choosing

$$\begin{aligned} \tau_i &= \sqrt{(a_i^{(i-1)} b_i + f_i^{(i-1)} g_i)^2 + h_i^2}, \\ \phi_i &= (a_i^{(i-1)} b_i + f_i^{(i-1)} g_i)/\tau_i, \\ \psi_i &= h_i/\tau_i, \end{aligned}$$

and forming $a^{(i)} = G_i a^{(i-1)}$ and $f^{(i)} = G_i f^{(i-1)}$.

After the $n-1$ Givens rotations have been applied

$$Q_2^T Q_1^T A = Q_2^T H = G_{n-1} \ldots G_1 H = R, \tag{3.24}$$

where $R$ is upper triangular. The strictly upper triangular part of $R$ is the strictly upper triangular part of $a^{(n-1)} b^T + f^{(n-1)} g^T$, where

$$a^{(n-1)} = G_{n-1} \ldots G_1 a \quad \text{and} \quad f^{(n-1)} = G_{n-1} \ldots G_1 f,$$

and the $i$th diagonal of $R$ is given by $\tau_i$ for $i = 1\colon n-1$ and the $n$th diagonal element is given by $\tau_n \equiv a_n^{(n-1)} b_n + f_n^{(n-1)} g_n$. Since $Q_2^T$ is a product of $n-1$ Givens rotations applied from top to bottom the structure of $Q_2^T$ is given by (3.5).

From (3.24), we have $A^{-1} = R^{-1} Q_2^T Q_1^T$. In order to compute $\|A^{-1}\|_1$ we first show how to find the lower triangular elements of $R^{-1} Q_2^T$ by using a similar approach to how the lower triangular elements of the inverse of a tridiagonal matrix are found in Section 3.2.1.

The $(i,j)$ element of $R^{-1} Q_2^T$ is

$$(R^{-1} Q_2^T)_{ij} = e_i^T R^{-1} Q_2^T e_j = u_j e_i^T R^{-1} v, \quad i \geq j,$$

where $e_i$ denotes the $i$th unit vector. If we let $w$ be the solution of the triangular system $Rw = v$ then

$$(R^{-1}Q_2^T)_{ij} = u_j w_i, \quad i \geq j.$$

In [6] it is shown that if the strictly upper triangular part of $R$ is the strictly upper triangular part of a rank-2 matrix then the triangular system $Rw = v$ can be solved in $O(n)$ operations. However this method would require products of $\psi_i$ in $u$ and $v$ which may cause underflow and overflow problems. Instead, as in Section 3.2.1 we scale $Rw = v$ using the diagonal matrix $D$ (3.4) in order to avoid products of $\psi_i$, to give $R'w' = v'$ where

$$R' = D^{-1}RD, \quad w' = D^{-1}w, \quad v' = D^{-1}v = [\phi_1, \ldots, \phi_{n-1}, 1]^T.$$

The scaled system can also be solved in $O(n)$ operations due to the structure of $D$. Given $R$, $D$ and $v'$, the following solves $R'w' = v'$ where $R' = D^{-1}RD$.

```
1  w'_n = v'_n/τ_n
2  s' = (-1)^{n-1} ψ_{n-1} w'_n [b_n  g_n]^T
3  w'_{n-1} = (v'_{n-1} - (-1)^{n-2} [a_{n-1}  f_{n-1}] s')/τ_{n-1}
4  for i = n-2: -1: 1
5        s' = (s' + (-1)^i w'_{i+1} [b_{i+1}  g_{i+1}]^T) ψ_i
6        w'_i = (v'_i - (-1)^{i-1} [a_i  f_i] s')/τ_i
7  end
```

(3.25)

Let

$$u' = [1, \phi_1, \ldots, \phi_{n-1}]^T.$$

Since $w = Dw'$ and $u = D^{-1}u'$, the diagonal elements of $R^{-1}Q_2^T$ are given by

$$u_i w_i = u'_i w'_i, \quad i = 1{:}\, n. \tag{3.26}$$

The strictly lower triangular elements are given by

$$u_j w_i = (-1)^{i+j-1} \psi_j \ldots \psi_{i-1} u'_j w'_i, \quad i > j. \tag{3.27}$$

The lower triangular part of $R^{-1}Q_2^T$ is therefore defined by the three vectors, $u'$, $w'$ and $\psi$.

The structure of $\mathrm{tril}(R^{-1}Q_2^T, 0)$ and $Q_1^T$ can now be exploited to find the $j$th absolute column sum of the strictly lower triangular part of $A^{-1} = R^{-1}Q_2^T Q_1^T$. Using (3.26), (3.27) and (3.23), the $(i,j)$, $i > j$, element of $A^{-1}$ is given by

$$\begin{aligned}(A^{-1})_{ij} &= \sum_{k=1}^{j} \left( (-1)^{k+i-1} \prod_{l=k}^{i-1} \psi_l \right) u_k' w_i' r_k s_j + \left( (-1)^{j+i} \prod_{l=j+1}^{i-1} \psi_l \right) u_{j+1}' w_i' t_j \\ &= \left( (-1)^{i+j} \prod_{l=j+1}^{i-1} \psi_l \right) w_i' \left( s_j \sum_{k=1}^{j} \left( (-1)^{j+k} \prod_{l=k}^{j} \psi_l \right) u_k' r_k + u_{j+1}' t_j \right).\end{aligned}$$

The $j$th absolute column sum of the strictly lower triangular part of $A^{-1}$ is therefore given by

$$\sigma_j = \left| \sum_{i=j+1}^{n} \left( (-1)^{j+i} \prod_{l=j+1}^{i-1} \psi_l \right) w_i' \right| \cdot \left| s_j \sum_{k=1}^{j} \left( (-1)^{k+j} \prod_{l=k}^{j} \psi_l \right) u_k' r_k + u_{j+1}' t_j \right|.$$

By considering

$$x_j' = \sum_{i=j+1}^{n} \left( (-1)^{j+i} \prod_{l=j+1}^{i-1} \psi_l \right) w_i'$$

and

$$y_j' = s_j \sum_{k=1}^{j} \left( (-1)^{k+j} \prod_{l=k}^{j} \psi_l \right) u_k' r_k + u_{j+1}' t_j,$$

separately, given $w', u', \psi, r, s$ and $t$ we can compute $\sigma_j$ for $j = 1\colon n-1$ in $O(n)$ operations as follows:

```
1  ψ = −ψ
2  y'_1 = ψ_1 u'_1 r_1
3  x'_n = |w'_n|
4  for i = 2: n − 1
5        y'_i = (y'_{i−1} + u'_i r_i) ψ_i
6        x'_{n−i} = |x'_{n−i+1} ψ_{n−i+1} + w'_{n−i+1}|
7  end
8  for i = 1: n − 1
9        σ_i = x'_i |y'_i s_i + u'_{i+1} t_i|
10 end
```
(3.28)

In order to find the $j$th absolute column sum of the upper triangular part of $A^{-1}$, $\widetilde{\sigma}_j$, we can consider $\widetilde{A} = JAJ$, where $J$ has only ones on the antidiagonal.

By swapping $p$ and $q$ with $y$ and $x$ respectively, reversing the order of the vectors $z$, $p$, $q$, $x$ and $y$, and setting $z_i = x_i y_i + z_i - p_i q_i$ for $i = 1\colon n$, we can repeat the above process on $\widetilde{A}$ to find the absolute columns sums of the strictly lower triangular part of $\widetilde{A}^{-1}$ and hence the absolute column sums of the strictly upper triangular part of $A^{-1}$.

We have shown how to find the strictly lower and strictly upper triangular parts of $A^{-1}$, and it remains to find the diagonal elements of $A^{-1}$. Given the strictly lower and strictly upper triangular parts of $A^{-1}$ it is possible to use $AA^{-1} = I$ to find the diagonal elements of $A^{-1}$ in $O(n)$ operations. However, this requires division by the diagonal elements of $A$, and the diagonal may contain zeros or elements close to zero that cause either breakdown of the algorithm or inaccurate results.

An alternative is to consider $RA^{-1} = Q_2^T Q_1^T$ and make use of

$$R(i,:)A^{-1}(:,i) = (Q_2^T Q_1^T)_{ii}.$$

The diagonal elements of $A^{-1}$ therefore satisfy

$$(A^{-1})_{ii} = \begin{cases} \frac{(Q_2^T Q_1^T)_{ii} - R(i,i+1:n)A^{-1}(i+1:n,i)}{\tau_i}, & i < n, \\ \frac{(Q_2^T Q_1^T)_{ii}}{\tau_i}, & i = n. \end{cases} \tag{3.29}$$

It is possible to divide by $\tau_i \equiv R(i,i)$ since $\tau_i \neq 0$ as $A$ is nonsingular. The structure of $Q_1^T$ and $Q_2^T$ is described in Proposition 3.3.1 and Theorem 3.2.1 respectively, and it is not difficult to check that given that we have used the code in (3.25) and (3.28), the following computes $c_i := (Q_2^T Q_1^T)_{ii}$ for $i < n$ in $O(n)$ operations.

```
1  s' = w_n'[b_n  g_n]
2  c_{n-1} = y_{n-1}' s' [a_{n-1}  f_{n-1}]^T
3  for i = n-2:-1:1
4        s' = psi_{i+1} s' + w_{i+1}' [b_{i+1}  g_{i+1}]
5        c_i = y_i' s' [a_i  f_i]^T
6  end
```
(3.30)

Similarly, by considering the structure of $R$ and the strictly lower triangular part of $A^{-1}$, we can compute $\widetilde{c}_i := R(i, i+1\colon n)A^{-1}(i+1\colon n, i)$ for $i \leq n$ in $O(n)$ operations as follows.

$$
\begin{array}{ll}
1 & \widetilde{c}_1 = u_1' a_1 \\
2 & \text{for } i = 2\colon n \\
3 & \qquad \widetilde{c}_i = \widetilde{c}_{i-1}\psi_{i-1} + u_i' a_i \\
4 & \text{end} \\
5 & \text{for } i = 1\colon n-1 \\
6 & \qquad \widetilde{c}_i = \widetilde{c}_i v_i s_i + \psi_i t_i \\
7 & \text{end} \\
8 & \widetilde{c}_n = \widetilde{c}_n v_n s_n
\end{array}
\tag{3.31}
$$

The absolute column sums and the absolute value of the diagonal of $A^{-1}$ can now be combined to give

$$\|A^{-1}\|_1 = \max_{i=1:n}(\sigma_i + \widetilde{\sigma}_i + |(A^{-1})_{ii}|).$$

The following algorithm computes $\|A^{-1}\|_1$ in $O(n)$ operations.

**Algorithm 3.3.2.** *Computes $\xi = \|A^{-1}\|_1$, where $A$ is an $n \times n$ dpss matrix given by* (3.20) *with $q_n \neq 0$ and $x_1 \neq 0$.*

```
1  Compute vectors in (3.21), (3.22) and (3.23) efficiently
2  for i = 1:n-1
3      (φ_i, ψ_i, τ_i) = Givens(a_i b_i + f_i g_i, h_i) and let G_i be given by (3.3)
4      a = G_i a, f = G_i f
5      u_{i+1} = φ_i, v_i = φ_i
6  end
7  τ_n = a_n b_n + f_n g_n
8  Solve R'w' = v' using the code in (3.25)
9  Compute σ_i for i < n using the code in (3.28)
10 Set σ̃ = σ and repeat above on Ã
11 Compute (Q_2^T Q_1^T)_{ii} using the code in (3.30)
12 Compute R(i, i+1 : n)A^{-1}(i+1 : n, i) using the code in (3.31)
13 Compute (A^{-1})_{ii} for i ≤ n using (3.29)
14 ξ = max_{i=1:n}(σ̃_i + σ_{n-i+1} + |(A^{-1})_{n-i+1,n-i+1}|)
```

**Cost**: Assuming the function *Givens* requires 6 operations, the total cost is $133n$ operations.

The restrictions $q_n \neq 0$ and $x_1 \neq 0$ in Algorithm 3.3.2 can easily be removed by considering the structure of $A^{-1}$. For example, if $q_i, \ldots, q_n$ are all zero, then the last $n-i+1$ columns of the strictly lower triangular part of $A^{-1}$ are zero and $(A^{-1})_{jj} = 1/z_i$ for $j = i\colon n$. Therefore, the absolute column sums of the strictly lower triangular part of $A^{-1}$ can be computed by applying lines 1–9 on $A(1\colon i-1, 1\colon i-1)$.

### 3.3.2 Numerical Experiments

To test Algorithm 3.3.2 we gave the defining vectors of a dpss matrix random elements in $[-1, 1]$. We tested the resulting dpss matrix and the matrices obtained by raising the components of the defining vectors to the powers 2 to 6 elementwise. This was repeated twenty times to give 120 test matrices with condition numbers varying from approximately $10^4$ to $5 \times 10^{25}$. The condition numbers of these test matrices were computed using Algorithm 3.3.2 and by forming the dpss matrices and using MATLAB's `cond`. The quantity $\beta = (\xi - \kappa_1(A))/\kappa_1(A)^2$ was computed, where $\xi$ denotes the condition number computed using Algorithm 3.3.2 and $\kappa_1(A)$ denotes the condition number computed using `cond`, and they are shown in Figure 3.1. We divide by $\kappa_1(A)^2$ since the condition number of computing $\kappa_1(A)$ is $\kappa_1(A)$ [25]. Thus we expect $\beta = O(u)$ for a forward stable method, where $u \approx 10^{-16}$ is the unit roundoff. The results show that Algorithm 3.3.2 performs in a forward stable way.

The test matrices described above are all full matrices. Diagonal-plus-semiseparable matrices with various components of the defining vectors set to zero were also tested for which accurate results were obtained.

The condition number of a general real matrix can be estimated in $O(n^2)$

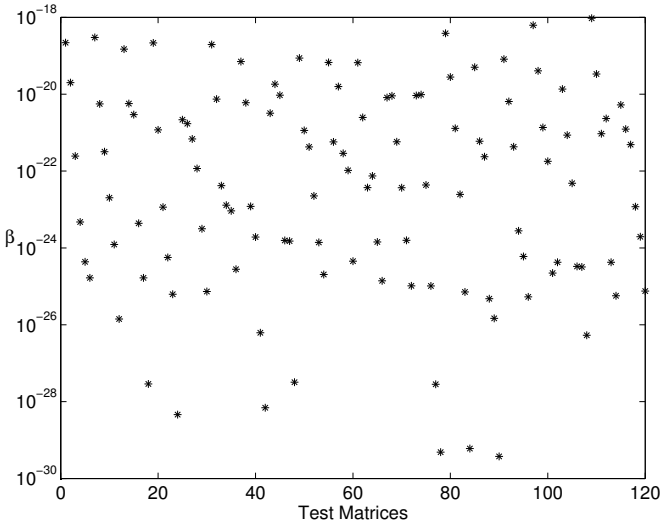


Figure 3.1: $\beta = (\xi - \kappa_1(A))/\kappa_1(A)^2$ for 120 test matrices $A \in \mathbb{R}^{100\times 100}$ with varying condition numbers.

operations using the LAPACK [3] routine `DLACON`. The algorithm estimates $\|B\|_1$ iteratively, for an arbitrary $B$, by computing the matrix-vector products $Bx$ and $B^Ty$ at each iteration for carefully chosen $x$ and $y$. No more than five iterations are usually required [30, Sec. 15.3]. Therefore if we have a factorization of $A$ such as an LU factorization or a QR factorization then we can form the matrix-vector products for $B = A^{-1}$ in $O(n^2)$ operations and hence estimate $\|A^{-1}\|_1$ in $O(n^2)$ operations.

In [6], given the defining vectors of a dpss matrix, $A$, the QR factorization of $A$ is considered to give an algorithm for solving the linear system $Ax = b$ in $O(n)$ operations. By storing the required vectors that define the QR factorizations of $A$ and $A^T$, the LAPACK algorithm can be adapted to estimate $\|A^{-1}\|_1$ in $O(n)$ operations. We emphasise that this method only *estimates* $\|A^{-1}\|_1$ and that inaccurate estimates can easily occur.

We have found experimentally that estimating the 1-norm of the inverse of a dpss matrix in this way requires two iterations, but this still requires

$177n$ operations which, is $44n$ more than that required for Algorithm 3.3.2. Table 3.5 shows the times taken in seconds to estimate $\|A^{-1}\|_1$ using an adapted version of the LAPACK algorithm for dpss matrices and the times taken to compute $\|A^{-1}\|_1$ using Algorithm 3.3.2. The algorithms were implemented using Fortran 95 and run on a 2010MHz AMD Athlon machine. The results show that Algorithm 3.3.2, which actually computes $\|A^{-1}\|_1$, is quickest.

Table 3.5: Time taken in seconds to compute the 1-norm of the inverse of a random $n \times n$ dpss matrix, for various $n$.

| Algorithm | $n = 5 \times 10^5$ | $n = 10^6$ | $n = 2 \times 10^6$ | $n = 5 \times 10^6$ |
|---|---|---|---|---|
| Algorithm 3.3.2 | 0.78 | 1.55 | 3.04 | 7.44 |
| LAPACK style | 1.15 | 2.23 | 4.52 | 10.83 |

## 3.4 Conclusion

We have presented two algorithms that compute the condition number of a tridiagonal matrix in $O(n)$ operations. The algorithms avoid underflow and overflow and do not require tests for degenerate cases as in [17]. The second algorithm is marginally slower than the quickest algorithm in [17], but is faster than the recommended algorithm in [17]. Our Fortran 77 implementations of these algorithms, which are presented in Appendix B, have been successfully tested using LAPACK testing codes. It, has been proposed that they be included in a future release of LAPACK.

An $O(n)$ algorithm to compute the condition number of a diagonal-plus-semiseparable matrix has also been given. Not only does this compute the condition number exactly, but it is also significantly quicker than a specialised implementation of the LAPACK condition number estimator.

# Chapter 4

# Efficient Algorithms for the Matrix Cosine and Sine

## 4.1 Introduction

The matrix exponential, undoubtedly the most-studied matrix function, provides the solution $y(t) = e^A y_0$ to the first order differential system $dy/dt = Ay$, $y(0) = y_0$, where $A \in \mathbb{C}^{n\times n}$ and $y \in \mathbb{C}^n$. Trigonometric matrix functions play a similar role in second order differential systems. For example, the problem

$$\frac{d^2y}{dt^2} + Ay = 0, \qquad y(0) = y_0, \quad y'(0) = y_0'$$

has solution[1]

$$y(t) = \cos(\sqrt{A}t)\, y_0 + \left(\sqrt{A}\right)^{-1} \sin(\sqrt{A}t)\, y_0', \tag{4.1}$$

where $\sqrt{A}$ denotes any square root of $A$. More general problems of this type, with a forcing term $f(t)$ on the right-hand side, arise from semidiscretisation of the wave equation and from mechanical systems without damping, and their solutions can be expressed in terms of integrals involving the sine and cosine

[1] This formula is interpreted for singular $A$ by expanding $\left(\sqrt{A}\right)^{-1} \sin(\sqrt{A}t)$ as a power series in $A$.

[44]. Despite the important role played by the matrix sine and cosine in these second order differential systems, their numerical computation has received relatively little attention. As well as methods for computing them individually, methods are needed for simultaneously computing the sine and cosine of the same matrix, as naturally arises in (4.1).

A general algorithm for computing the matrix cosine that employs rational approximations and the double-angle formula $\cos(2A) = 2\cos^2(A) - I$ was proposed by Serbin and Blalock [45]. Higham and Smith [34] developed a particular version of this algorithm based on Padé approximation and supported by truncation and rounding error analysis. In this work we revisit the algorithm of Higham and Smith, making several improvements to increase both its efficiency and its accuracy and adapting it to compute $\cos(A)$ and $\sin(A)$ together.

First, we state the original algorithm [34, Alg. 6.1]. This algorithm, and all those discussed here, are intended for use in IEEE double precision arithmetic [35], for which the unit roundoff $u = 2^{-53} \approx 1.11 \times 10^{-16}$.

**Algorithm 4.1.1.** *Given a matrix $A \in \mathbb{C}^{n\times n}$ this algorithm approximates $X = \cos(A)$.*

1 Find the smallest nonnegative integer $m$ so that $2^{-m}\|A\|_\infty \leq 1$.
2 $C_0 = r_{88}(2^{-m}A)$, where $r_{88}(x)$ is the [8/8] Padé approximant to $\cos(x)$.
3 for $i = 0\colon m-1$
4 $\quad C_{i+1} = 2C_i^2 - I$
5 end
6 $X = C_m$

**Cost**: $(4 + \mathrm{ceil}(\log_2 \|A\|_\infty))M + D$, where $M$ denotes a matrix multiplication and $D$ the solution of a linear system with $n$ right-hand side vectors.

The algorithm can be explained as follows. Line 1 determines the scaling needed to reduce the $\infty$-norm of $A$ to 1 or less. Line 2 computes the [8/8] Padé

approximant of the scaled matrix; it is evaluated by the technique described in Section 4.2 (cf. (4.7)) at a cost of $4M + D$. The loop beginning at line 3 uses the double-angle formula $\cos(2A) = 2\cos(A)^2 - I$ to undo the effect of the scaling.

Higham and Smith [34] show that

$$\frac{\|\cos(A) - r_{88}(A)\|_\infty}{\|\cos(A)\|_\infty} \leq 3.26 \times 10^{-16} \approx 3u \qquad \text{for } \|A\|_\infty \leq 1. \tag{4.2}$$

Hence in Algorithm 4.1.1, $r_{88}(2^{-m}A)$ approximates $C_0 = \cos(2^{-m}A)$ to essentially full machine accuracy.

Algorithm 4.1.1 can optionally make use of preprocessing, which is implemented in the next algorithm.

**Algorithm 4.1.2.** *Given a matrix $A \in \mathbb{C}^{n\times n}$ this algorithm computes $X = \cos(A)$ by preprocessing $A$ and then invoking a given algorithm for computing $\cos(A)$.*

1 $A \leftarrow A - \pi q I$, where $q$ is whichever of 0, floor($\mu$) and ceil($\mu$) yields the smaller value of $\|A - \pi q I\|_\infty$, where $\mu = \text{trace}(A)/(n\pi)$.
2 $B = D^{-1}AD$, where $D$ balances $A$.
3 if $\|B\|_\infty < \|A\|_\infty$, $A = B$, end
4 Apply the given algorithm to compute $C = \cos(A)$.
5 $X = (-1)^q C$
6 if balancing was performed, $X \leftarrow DXD^{-1}$, end

Lines 1-3 carry out preprocessing prior to the main computations; they apply a similarity transformation and a shift in an attempt to reduce the norm. Lines 5 and 6 undo the effect of the preprocessing. See [34] for an explanation of the preprocessing.

The impetus for this work comes from two observations. First, the analysis of Higham and Smith focuses on the [8/8] Padé approximant, but the use of an approximant of a different, $A$-dependent degree could potentially yield a

more efficient algorithm. The recent work of Higham [32] on the scaling and squaring method for the matrix exponential shows how to choose the degree of the Padé approximant and the norm of the scaled matrix at which the approximant is evaluated in order to obtain an optimally efficient algorithm, and the same approach is applicable to the double-angle algorithm for the cosine. The second relevant observation is that the double-angle steps in Algorithm 4.1.1 can potentially magnify both truncation and rounding errors substantially, so reducing the number of such steps (while not sacrificing the efficiency of the whole algorithm) could bring an important improvement in accuracy. Indeed it is shown in [34] that the computed $\widehat{C}_i =: C_i + E_i$ satisfies

$$\begin{aligned}\|E_i\|_\infty \leq (4.1)^i \|E_0\|_\infty \|C_0\|_\infty \|C_1\|_\infty \dots \|C_{i-1}\|_\infty \\ + \gamma_{n+1} \sum_{j=0}^{i-1} 4.1^{i-j-1} (2.21\|C_j\|_\infty^2 + 1) \|C_{j+1}\|_\infty \dots \|C_{i-1}\|_\infty, \quad (4.3)\end{aligned}$$

where $\gamma_k = ku/(1-ku)$, which warns of error growth exponential in the number of double-angle steps, but Algorithm 4.1.1 does not attempt to minimise the number of such steps.

In this work we show how to choose the degree of the Padé approximant to minimise the computational effort while at the same time (approximately) minimising the number of double-angle steps, and where minimisation is subject to retaining numerical stability in evaluation of the Padé approximant. We also show how to exploit the fact that the cosine is an even function to reduce the work, possibly by a large amount.

In Section 4.2 we develop an improved version of Algorithm 4.1.1 that incorporates these ideas. In Section 4.3 we argue that imposing an absolute, rather than relative, error criterion on the Padé approximant leads to a more efficient algorithm whose accuracy is in general no worse. The numerical experiments of Section 4.4 compare Algorithm 4.1.1 with the two new algorithms derived in this chapter and also with MATLAB's `funm` applied to the cosine.

These sections concentrate on the cosine. There is no natural analogue of Algorithm 4.1.1 for the sine, because the corresponding double-angle recurrence $\sin(2A) = 2\sin(A)\cos(A)$ would require cosines. However, computing the sine reduces to computing the cosine through $\sin(A) = \cos(A - \frac{\pi}{2}I)$.

Building on the new algorithms for the cosine, in Section 4.5 we develop an algorithm for simultaneously computing $\cos(A)$ and $\sin(A)$ at lower cost than if they were computed independently, which is useful when evaluating (4.1), for example. Concluding remarks are given in Section 4.6.

Throughout this chapter an unsubscripted norm denotes an arbitrary subordinate matrix norm.

## 4.2 An Algorithm with Variable Degree Padé Approximants

We denote by $r_m(x) = p_m(x)/q_m(x)$ an $[m/m]$ Padé approximant of a given function $f(x)$. By definition, $p_m$ and $q_m$ are polynomials in $x$ of degree at most $m$ and

$$f(x) - r_m(x) = O(x^{2m+1}).$$

We will normalise so that $p_m$ and $q_m$ have no common zeros and $q_m(0) = 1$. For later reference we write

$$p_m(x) = \sum_{i=0}^{m} a_i x^i, \qquad q_m(x) = \sum_{i=0}^{m} b_i x^i. \tag{4.4}$$

As discussed in [34], it is not known whether Padé approximants of $\cos(x)$ exist for all $m$, though formulae of Magnus and Wynn [39] are available that give the coefficients of $p_m$ and $q_m$ in terms of ratios of determinants of matrices whose entries involve binomial coefficients. Since cos is an even function we need consider only even degrees $2m$. Both $p_{2m}$ and $q_{2m}$ are even polynomials

and

$$\cos(x) - r_{2m}(x) = O(x^{4m+2}).$$

Our first task is to bound the truncation error, which has the form

$$\cos(A) - r_{2m}(A) = \sum_{i=2m+1}^{\infty} c_{2i} A^{2i}.$$

Hence

$$\| \cos(A) - r_{2m}(A) \| \leq \sum_{i=2m+1}^{\infty} |c_{2i}| \theta^{2i}, \tag{4.5}$$

where

$$\theta = \theta(A) = \|A^2\|^{1/2}.$$

Note that we have expressed the bound in terms of $\|A^2\|^{1/2}$ instead of the (no smaller) quantity $\|A\|$. The reason is that $\|A^2\|^{1/2} \ll \|A\|$ is possible for nonnormal $A$. Since our algorithm will require the matrix $A^2$ to be computed, it makes sense to use knowledge of its norm in the derivation; this was not done in [34] and so is one way in which we gain an improvement over Algorithm 4.1.1.

It is easy to see that

$$\| \cos(A) \| \geq 1 - \frac{\|A^2\|}{2!} - \frac{\|A^2\|^2}{4!} - \cdots = 1 - (\cosh(\|A^2\|^{1/2}) - 1) = 2 - \cosh(\theta).$$

Combining this bound with (4.5), we conclude that

$$\frac{\| \cos(A) - r_{2m}(A) \|}{\| \cos(A) \|} \leq \frac{\sum_{i=2m+1}^{\infty} |c_{2i}| \theta^{2i}}{2 - \cosh(\theta)} \quad \text{for } \theta < \cosh^{-1}(2) \approx 1.317. \tag{4.6}$$

To design the algorithm we need to know for each $m$ how small $\theta = \|A^2\|^{1/2}$ must be in order for $r_{2m}$ to deliver the required accuracy. Adopting the approach used by Higham [32] for the exponential, we therefore determine the largest value of $\theta$, denoted by $\theta_{2m}$, such that the relative error bound in (4.6) does not exceed $u$. To do so we compute the $c_{2i}$ symbolically and evaluate the bound (4.6) in 250 decimal digit arithmetic, summing the first 150 terms of the series, all with MATLAB's Symbolic Math Toolbox. We use a zero-finder

Table 4.1: Maximum value $\theta_{2m}$ of $\theta = \|A^2\|^{1/2}$ such that the relative error bound (4.6) does not exceed $u = 2^{-53}$.

| $2m$ | 2 | 4 | 6 | 8 | 10 | 12 | 14 |
|---|---|---|---|---|---|---|---|
| $\theta_{2m}$ | 6.1e-3 | 1.1e-1 | 4.3e-1 | 9.5e-1 | 1.315 | 1.317 | 1.317 |

to find $\theta_{2m}$, obtaining the values shown in Table 4.1. We see that $\theta_{2m}$ rapidly approaches $\cosh^{-1}(2)$ as $m$ increases: $\theta_{14}$ and $\theta_{12}$ differ by about $10^{-9}$.

When we rerun the computation with $2m = 8$, aiming for a relative error bound $3.26 \times 10^{-16}$, we find that $\theta_8 = 1.005$, which shows that the bound (4.2) is close to optimal (modulo its use of $\|A\|$ in place of $\|A^2\|^{1/2}$).

Now we need to determine the cost of evaluating $r_{2m}$. Given the absence of any convenient continued fraction or partial fraction forms, we will explicitly evaluate $p_{2m}$ and $q_{2m}$ and then solve the multiple right-hand side system $q_{2m}r_{2m} = p_{2m}$. As noted in Section 1.7.1, the most efficient method of evaluating a single polynomial is the Paterson Stockmeyer method [43]. To evaluate both $p_{2m}$ and $q_{2m}$, the most efficient scheme we have found is to treat the two polynomials as degree $m$ polynomials in $A^2$ and apply the Paterson–Stockmeyer method, adapted for the simultaneous evaluation of two polynomials of the same argument and degree as suggested by Higham [29]. Of equal cost for $8 \le 2m \le 28$ are the schemes of the form illustrated for $2m = 12$ by

$$\begin{aligned} &A_2 = A^2, \quad A_4 = A_2^2, \quad A_6 = A_2A_4, \\ &p_{12} = a_0I + a_2A_2 + a_4A_4 + a_6A_6 + A_6(a_8A_2 + a_{10}A_4 + a_{12}A_6), \\ &q_{12} = b_0I + b_2A_2 + b_4A_4 + b_6A_6 + A_6(b_8A_2 + b_{10}A_4 + b_{12}A_6). \end{aligned} \tag{4.7}$$

Table 4.2 summarises the cost of evaluating $p_{2m}$ and $q_{2m}$ for $2m = 2\colon 2\colon 30$.

In view of Table 4.1 we can restrict to $2m \le 12$, since $\theta_{14}$ is only slightly larger than $\theta_{12}$. Since Table 4.2 shows that $r_{12}$ can be evaluated at the same cost as the less accurate $r_{10}$, we can remove $2m = 10$ from consideration. Hence

Table 4.2: Number of matrix multiplications $\pi_{2m}$ required to evaluate $p_{2m}(A)$ and $q_{2m}(A)$.

| $2m$ | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 | 26 | 28 | 30 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\pi_{2m}$ | 1 | 2 | 3 | 4 | 5 | 5 | 6 | 6 | 7 | 7 | 8 | 8 | 9 | 9 | 9 |

we need consider only $2m = 2, 4, 6, 8, 12$.

If $\theta = \|A^2\|^{1/2} \leq \theta_{2m}$, for $2m = 2, 4, 6, 8$ or 12, then we should take $r_{2m}(A)$ with the smallest such $m$ as our approximation to $\cos(A)$. Otherwise, we will need to scale: we simply divide $A$ by $2^s$, with $s$ chosen minimally so that $\|(2^{-s}A)^2\|_\infty^{1/2} \leq \theta_{2m}$ for some $m$, with $2m = 8$ and $2m = 12$ being the only possibilities (since $\theta_6 < \theta_{12}/2$, $2m = 6$ offers no computational saving over $2m = 12$). This strategy minimises the number of double-angle steps, with their potential error magnification, while at the same time minimising the total work.

We now need to consider the effects of rounding errors on the evaluation of $r_{2m}$. Consider, first, the evaluation of $p_{2m}$ and $q_{2m}$, and assume initially that $A^2$ is evaluated exactly. Let $g_{2m}(A^2)$ denote either of the even polynomials $p_{2m}(A)$ and $q_{2m}(A)$. It follows from a general result in [32, Thm. 2.2] that

$$\|g_{2m}(A^2) - fl(g_{2m}(A^2))\| \leq \widetilde{\gamma}_{mn}\widetilde{g}_{2m}(\|A^2\|), \tag{4.8}$$

where $\widetilde{g}_{2m}$ denotes $g_{2m}$ with its coefficients replaced by their absolute values. We have determined numerically that $\widetilde{g}_{2m}(\|A\|^2) \leq 2$ for $\theta(A) \leq \theta_{2m}$ and $2m \leq 16$, so the bound (4.8) is suitably small. However, when we take into account the error in forming $A^2$ we find that the bound (4.8) is multiplied by a term that is approximately $\mu(A) = \||A|^2\|/\|A^2\| \geq 1$. The quantity $\mu$ can be arbitrarily large. However, $\mu$ is large precisely when basing the algorithm on $\theta(A)$ rather than $\|A\|$ produces a smaller $s$, so potentially increased rounding errors in the evaluation of $p_{2m}$ and $q_{2m}$ are balanced by potentially decreased

Table 4.3: Upper bound for $\kappa(q_{2m}(A))$ when $\theta \le \theta_{2m}$, based on (4.9) and (4.10), where the $\theta_{2m}$ are given in Table 4.1.

| $2m$ | 2 | 4 | 6 | 8 | 12 |
|---|---|---|---|---|---|
| Bound | 1.0 | 1.0 | 1.0 | 1.0 | 1.1 |

error propagation in the double angle phase.

Since we obtain $r_{2m}$ by solving a linear system with coefficient matrix $q_{2m}(A)$, we require $q_{2m}(A)$ to be well conditioned to be sure that the system is solved accurately. From (4.4), we have

$$\|q_{2m}(A)\| \le \sum_{k=0}^{m} |b_{2k}|\theta^{2k}. \tag{4.9}$$

Using the inequality $\|(I+E)^{-1}\| \le (1-\|E\|)^{-1}$ for $\|E\| < 1$ gives

$$\|q_{2m}(A)^{-1}\| \le \frac{1}{|b_0| - \|\sum_{k=1}^{m} b_{2k}A^{2k}\|} \le \frac{1}{|b_0| - \sum_{k=1}^{m} |b_{2k}|\theta^{2k}}. \tag{4.10}$$

Table 4.3 tabulates the bound for $\kappa(q_{2m}(A)) = \|q_{2m}(A)\|\|q_{2m}(A)^{-1}\|$ obtained by combining (4.9) and (4.10). It shows that $q_{2m}$ is well conditioned for all the $m$ of interest.

The algorithm that we have derived is as follows.

**Algorithm 4.2.1.** *Given a matrix $A \in \mathbb{C}^{n\times n}$ this algorithm approximates $C = \cos(A)$. It uses the constants $\theta_{2m}$ given in Table* 4.1. *The matrix $A$ can optionally be preprocessed using Algorithm* 4.1.2.

```
1  B = A^2
2  θ = ||B||_∞^{1/2}
3  for d = [2 4 6 8 12]
4      if θ ≤ θ_d
5          C = r_d(A) % Compute Padé approximant, making use of B.
6          quit
7      end
8  end
```

9 $s = \text{ceil}(\log_2(\theta/\theta_{12}))$ % Find minimal integer $s$ such that $2^{-s}\theta \leq \theta_{12}$.
10 $B \leftarrow 4^{-s}B$
11 if $\|B\|_\infty^{1/2} \leq \theta_8$, $d = 8$, else $d = 12$, end
12 $C = r_d(2^{-s}A)$ % Compute Padé approximant, making use of $B = (2^{-s}A)^2$.
13 for $i = 1\colon s$
14 $\quad C \leftarrow 2C^2 - I$
15 end

**Cost**: $\big(\pi_d + \text{ceil}(\log_2(\|A\|_\infty/\theta_d))\big)M + D$, where $d$ is the degree of Padé approximant used and $\theta_d$ and $\pi_d$ are tabulated in Tables 4.1 and 4.2, respectively.

To summarise, Algorithm 4.2.1 differs from Algorithm 4.1.1 in two main ways:

1. It supports variable degree Padé approximation, with the degree chosen to minimise both the work and the number of double-angle steps.

2. It bases its decisions on $\|A^2\|_\infty^{1/2}$ rather than $\|A\|_\infty$ and so uses truncation error estimates that are potentially much sharper, though the bounds for the effect of rounding errors on the evaluation of the Padé approximant can be larger.

Note that Algorithm 4.2.1 requires as input only $A^2$, not $A$. Consequently, if it is used to compute the cosine term in (4.1) there is no need for a square root to be computed.

## 4.3 Absolute Error-Based Algorithm

Algorithm 4.2.1 uses a Padé approximant of maximal degree 12. The limitation on degree comes from requiring the relative error bound (4.6) to be no larger than $u$: the need to ensure $\cos(A) \neq 0$ enforces the restriction $\theta < \cosh^{-1}(2)$, which makes higher degree Padé approximants uneconomical. If this restriction

could be removed then larger degrees, which would allow fewer double-angle steps and hence potentially more accurate results, would be competitive in cost. Imposing an absolute error bound on the Padé approximant achieves this goal, and it can be justified with the aid of the error bound (4.3) for the computed $\widehat{C}_i =: C_i + E_i$.

In both Algorithm 4.1.1 and Algorithm 4.2.1, $C_0$ is a Padé approximant evaluated at $2^{-m}A$, and from $\|C_0\|_\infty = \|\cos(2^{-m}A)\|_\infty < \cosh(\theta(2^{-m}A))$ we have $\|C_0\|_\infty < \cosh(1) \approx 1.54$ in Algorithm 4.1.1 and $\|C_0\|_\infty \leq \cosh(\theta_{12}) \approx 2$ in Algorithm 4.2.1. Hence, ignoring the rounding errors in evaluating the Padé approximant, we have $\|E_0\|_\infty \lesssim u\|C_0\|_\infty \lesssim u$, and (4.3) can be written

$$\begin{aligned}\|E_m\|_\infty \lesssim\ & (4.1)^m u\|C_0\|_\infty\|C_1\|_\infty \dots \|C_{m-1}\|_\infty \\ & + \gamma_{n+1}\sum_{j=0}^{m-1} 4.1^{m-j-1}(2.21\|C_j\|_\infty^2 + 1)\|C_{j+1}\|_\infty \dots \|C_{m-1}\|_\infty \end{aligned} \quad (4.11)$$

But this is exactly the form that (4.3) takes with an absolute error bound $\|E_0\| \leq u$. Therefore comparing the effect of absolute and relative bounds on $C_0$ reduces to comparing the norms of the matrices $C_0$, ..., $C_{m-1}$ in the two cases. This is difficult in general, because these matrices depend on the choice of scaling and on $m$. Since the aim of using an absolute criterion is to allow the norm of the scaled matrix to be larger, we can expect an upper bound for $\|C_0\|$ to be larger in the absolute case. But if the absolute criterion permits fewer double-angle steps (a smaller $m$) then, as is clear from (4.11), significant gains in accuracy could accrue. In summary, the error analysis provides support for the use of an absolute error criterion if $\|C_0\|$ is not too large. We now develop an algorithm based on an absolute error bound.

Define $\theta_{2m}$ to be the largest value of $\theta$ such that the absolute error bound in

$$\|\cos(A) - r_{2m}(A)\| \leq \sum_{i=2m+1}^{\infty} |c_{2i}|\theta^{2i} \quad (4.12)$$

Table 4.4: Maximum value $\theta_{2m}$ of $\theta$ such that the absolute error bound (4.12) does not exceed $u = 2^{-53}$.

| $2m$ | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 |
|---|---|---|---|---|---|---|---|---|
| $\theta_{2m}$ | 6.1e-3 | 0.11 | 0.43 | 0.98 | 1.7 | 2.6 | 3.6 | 4.7 |
| $2m$ | 18 | 20 | 22 | 24 | 26 | 28 | 30 | |
| $\theta_{2m}$ | 5.9 | 7.1 | 8.3 | 9.6 | 10.9 | 12.2 | 13.6 | |

Table 4.5: Upper bound for $\kappa(q_{2m}(A))$ when $\theta \le \theta_{2m}$, based on (4.9) and (4.10), where the $\theta_{2m}$ are given in Table 4.4. Bound does not exist for $2m \ge 26$.

| $2m$ | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bound | 1.0 | 1.0 | 1.0 | 1.0 | 1.1 | 1.2 | 1.4 | 1.8 | 2.4 | 3.5 | 7.0 | 9.0e1 |

(a restatement of (4.5)) does not exceed $u$. Using the same method of determining the $\theta_{2m}$ as in the previous section we find the values listed in Table 4.4. The corresponding bounds for the condition number of $q_{2m}$, which are finite only for $2m \le 24$, are given in Table 4.5.

Now we consider the choice of $m$. In view of Table 4.5, we will restrict to $2m \le 24$. Table 4.6, which concerns error bounds for the evaluation of $p_{2m}$ and $q_{2m}$, as discussed in the previous section, suggests further restricting $2m \le 20$, say. From Table 4.2 it is then clear that we need consider only $2m = 2, 4, 6, 8, 12, 16, 20$. Recall that dividing $A$ (and hence $\theta$) by 2 results in one

Table 4.6: Upper bounds for $\|\widetilde{p}_{2m}\|_\infty$ and $\|\widetilde{q}_{2m}\|_\infty$ for $\theta \le \theta_{2m}$.

| $2m$ | 2 | 4 | 6 | 8 | 10 | 12 |
|---|---|---|---|---|---|---|
| $\|\widetilde{p}_{2m}\|_\infty$ | 1.0 | 1.0 | 1.1 | 1.5 | 2.7 | 6.2 |
| $\|\widetilde{q}_{2m}\|_\infty$ | 1.0 | 1.0 | 1.0 | 1.0 | 1.1 | 1.1 |
| $2m$ | 14 | 16 | 18 | 20 | 22 | 24 |
| $\|\widetilde{p}_{2m}\|_\infty$ | 1.6e1 | 4.3e1 | 1.2e2 | 3.7e2 | 1.2e3 | 3.7e3 |
| $\|\widetilde{q}_{2m}\|_\infty$ | 1.2 | 1.3 | 1.4 | 1.6 | 1.7 | 2.0 |

Table 4.7: Logic for choice of scaling and Padé approximant degree. Assuming $A$ has already been scaled, if necessary, so that $\theta \le \theta_{20} = 7.1$, further scaling should be done to bring $\theta$ within the range for the indicated value of $d$.

| Range of $\theta$ | $d$ |
|---|---|
| $[0, \theta_{16}] = [0, 4.7]$ | smallest $d \in \{2, 4, 6, 8, 12, 16\}$ such that $\theta \le \theta_d$ |
| $(\theta_{16}, 2\theta_{12}] = (4.7, 5.2]$ | 12 (scale by 1/2) |
| $(2\theta_{12}, \theta_{20}] = (5.2, 7.1]$ | 20 (no scaling) |

extra matrix multiplication in the double-angle phase, whereas for $\theta \le \theta_{2m}$ the cost of evaluating the Padé approximant increases by one matrix multiplication with each increase in $m$ in our list of considered values. Since the numbers $\theta_{12}$, $\theta_{16}$, $\theta_{20}$ differ successively by less than a factor 2, the value of $m$ that gives the minimal work depends on $\theta$. For example, if $\theta = 7$ then $d = 20$ is best, because nothing would be gained by a further scaling by 1/2, but if $\theta = 5$ then scaling by 1/2 enables us to use $d = 12$, and the whole computation then requires one less matrix multiplication than if we immediately applied $d = 20$. Table 4.7 summarises the relevant logic. The tactic, then, is to scale so that $\theta \le \theta_{20}$ and to scale further only if a reduction in work is achieved.

We find, computationally, that with this scaling strategy, $\|C_0\|_\infty \le 583$. Since this bound is not too much larger than 1, the argument at the beginning of this section provides justification for the following algorithm.

**Algorithm 4.3.1.** *Given a matrix $A \in \mathbb{C}^{n\times n}$ this algorithm approximates $C = \cos(A)$. It uses the constants $\theta_{2m}$ given in Table 4.4. The matrix $A$ can optionally be preprocessed using Algorithm 4.1.2.*

```
1  B = A^2
2  θ = ||B||_∞^{1/2}
3  for d = [2 4 6 8 12 16]
4      if θ ≤ θ_d
5          C = r_d(A) % Compute Padé approximant, making use of B.
6          quit
```

```
    end
end
s = ceil(log2(θ/θ20)) % Find minimal integer s such that 2^{-s}θ ≤ θ20.
Determine optimal d from Table 4.7 (with θ ← 2^{-s}θ) and
increase s as necessary.
B ← 4^{-s}B
C = r_d(2^{-s}A) % Compute Padé approximant, making
use of B = (2^{-s}A)^2.
for i = 1: s
    C ← 2C^2 − I
end
```

**Cost**: $\bigl(\pi_d + \mathrm{ceil}(\log_2(\|A\|_\infty/\theta_d))\bigr)M + D$, where $d$ is the degree of Padé approximant used and $\theta_d$ and $\pi_d$ are tabulated in Tables 4.4 and 4.2, respectively.

Algorithm 4.3.1 allows the norm $\|(2^{-s}A)^2\|_\infty^{1/2}$ for the scaled matrix $2^{-s}A$ to be as large as 7.1, compared with just 1.3 for Algorithm 4.2.1.

## 4.4 Numerical Experiments

Testing of Algorithms 4.1.1, 4.2.1, and 4.3.1 was performed in MATLAB 7 in IEEE double precision arithmetic. We used a set of 54 test matrices that includes 50 $10 \times 10$ matrices obtained from the function `matrix` in the Matrix Computation Toolbox [27] (which includes test matrices from MATLAB itself), together with the four test matrices from [34]. The norms of these matrices range from order 1 to $10^7$, though more than half have $\infty$-norm 10 or less. For comparison, we also applied MATLAB's `funm` function (invoked as `funm(A,@cos)`), which implements the Schur–Parlett method [16]. This method uses Taylor series evaluations of any diagonal Schur blocks of size greater than 1. It requires roughly between $28n^3$ flops and $n^4/3$ flops, so is significantly more expensive than Algorithms 4.1.1, 4.2.1, and 4.3.1 except, possibly, when $\|A^2\|_\infty^{1/2}$ is large: say of order $10^3$.

We evaluated the relative error

$$\frac{\|\widehat{C} - C\|_\infty}{\|C\|_\infty},$$

where $\widehat{C}$ is the computed approximation to $C$, and the exact $C = \cos(A)$ is computed in 50 significant decimal digit arithmetic using MATLAB's Symbolic Math Toolbox. The algorithms were applied both with and without preprocessing. The results for no preprocessing are shown in Figures 4.1; those for preprocessing are very similar so are omitted. The solid line is the unit roundoff multiplied by an estimate of the relative condition number

$$\mathrm{cond}(A) = \lim_{\epsilon \to 0} \max_{\|E\|_2 \le \epsilon \|A\|_2} \frac{\|\cos(A+E) - \cos(A)\|_2}{\epsilon \|\cos(A)\|_2}.$$

We compute such an estimate using the finite-difference power method of Kenney and Laub [37] as described in Section 1.7.2. A method that is forward stable should produce errors not lying far above this line on the graph. Figure 4.2 shows performance profile curves for the four solvers. For a given $\alpha$ on the $x$-axis, the $y$ coordinate of the corresponding point on the curve is the probability that the method in question has an error within a factor $\alpha$ of the smallest error over all the methods on the given test set.

The results show a clear ordering of the methods for this set of test problems, with Algorithm 4.3.1 in first place, followed by `funm`, Algorithm 4.2.1, and finally Algorithm 4.1.1.

The mean of the total number of matrix multiplications and multiple right-hand side linear system solves over the test set is 10, 9.1 and 8.6 for Algorithms 4.1.1, 4.2.1 and 4.3.1, respectively, without preprocessing, and 9.8, 8.9 and 8.4 with preprocessing. For the involutory matrix `gallery('invol',8)*8*pi` from [34], Algorithm 4.1.1 requires 29 multiplies and solves, versus only 10 for Algorithms 4.2.1 and 4.3.1.

MATLAB's `funm` is generally competitive in accuracy with Algorithm 4.3.1. The worst case for `funm`—the matrix giving error about $10^{-10}$ on the left of

Figure 4.1—is the Forsythe matrix, which is a $10^{-8}$ perturbation of a Jordan block. The computed eigenvalues lie approximately on a circle, and this is known to be a difficult case for `funm` [16]. Increasing the blocking tolerance, through the call `funm(A,@cos,struct('TolBlk',0.2))` results in an accurate evaluation.

We repeated the experiment with every matrix scaled so that $\|A\|_\infty = 25$. The results without preprocessing are shown in Figure 4.3; those with preprocessing are very similar, with just a modest reduction of up to a factor 3 or so of the maximum and mean error for Algorithms 4.1.1, 4.2.1 and 4.3.1. The performance profile is shown in Figure 4.4. Clearly the (generally) larger norm causes difficulty for all the methods, but much less so for Algorithm 4.3.1 than for Algorithms 4.1.1 and 4.2.1. In this case, the means costs are 10, 9.4 and 9.1 without preprocessing and 9.6, 9.1 and 8.6 with preprocessing.

## 4.5 Computing the Sine and Cosine of a Matrix

Suppose now that we wish to compute both $\sin(A)$ and $\cos(A)$. Since $\cos(A) = (e^{iA} + e^{-iA})/2$ and $\sin(A) = (e^{iA} - e^{-iA})/(2i)$, we can obtain both functions from two matrix exponential evaluations. However, when $A$ is real the arguments of the exponential are complex, so this approach will not be competitive in cost even with computing $\sin(A)$ and $\cos(A)$ separately. A further disadvantage is that these formulas can suffer badly from cancellation in floating point arithmetic, as shown in [34].

We will develop an analogue of Algorithm 4.3.1 that scales $A$ by a power of 2, computes Padé approximants to both the sine and cosine of the scaled matrix, and then applies the double-angle formulas $\cos(2A) = 2\cos^2(A) - I$

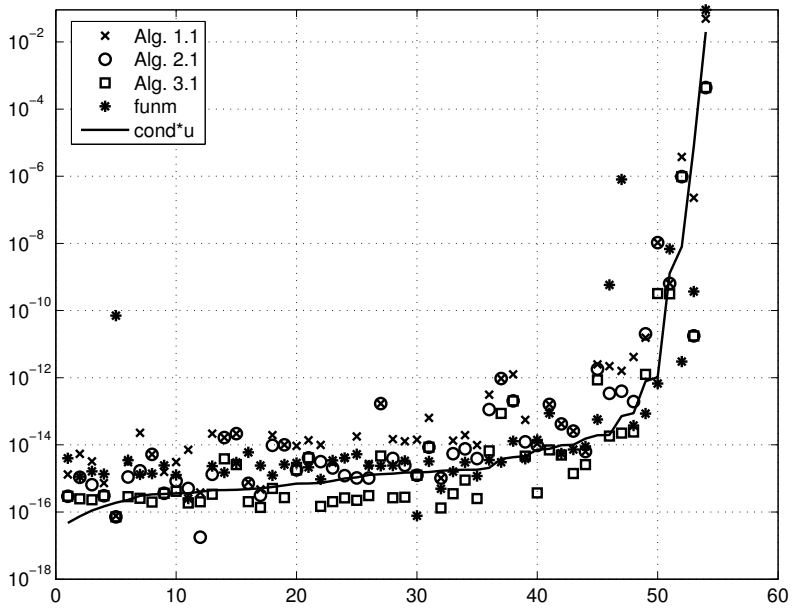


Figure 4.1: Errors for Algorithms 4.1.1, 4.2.1, and 4.3.1 without preprocessing.

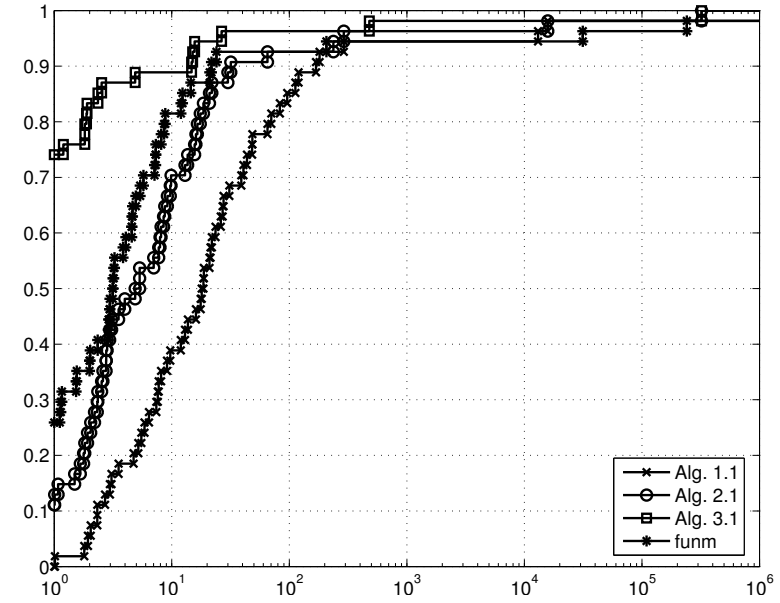


Figure 4.2: Performance profile for the four methods, without preprocessing, on the test set.

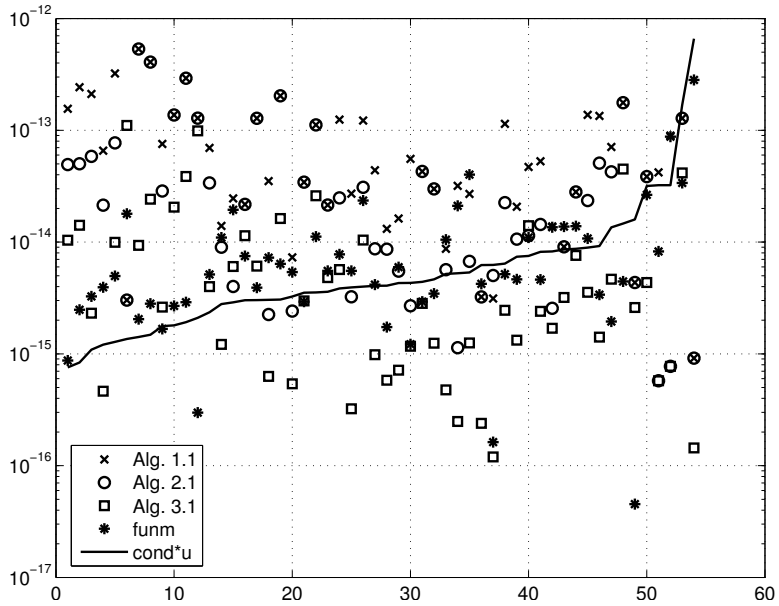


Figure 4.3: Errors for Algorithms 4.1.1, 4.2.1, and 4.3.1 without preprocessing on matrices scaled so that $\|A\|_\infty = 25$.

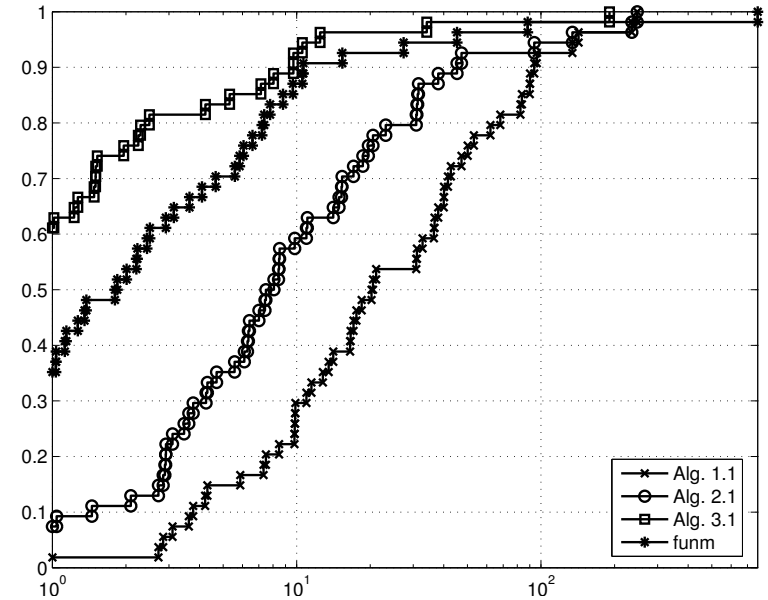


Figure 4.4: Performance profile for the four methods, without preprocessing, on the test set with matrices scaled so that $\|A\|_\infty = 25$.

Table 4.8: Maximum value $\beta_m$ of $\|A\|_\infty$ such that the relative error bound (4.13) does not exceed $u = 2^{-53}$.

| $m$ | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\beta_m$ | 1.4e-3 | 1.8e-2 | 6.4e-2 | 1.7e-1 | 0.32 | 0.56 | 0.81 | 1.2 | 1.5 | 2.0 | 2.3 | |
| $m$ | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
| $\beta_m$ | 2.9 | 3.3 | 3.9 | 4.4 | 5.0 | 5.5 | 6.2 | 6.7 | 7.4 | 7.9 | 8.7 | 9.2 |

and $\sin(2A) = 2\sin(A)\cos(A)$. Computational savings are possible in the evaluation of the Padé approximants and in the double-angle recurrences by re-using the cos terms.

Let us denote the $[m/m]$ Padé approximant to the sine function by $\widetilde{r}_m(x) = \widetilde{p}_m(x)/\widetilde{q}_m(x)$. Then the error in $\widetilde{r}_m$ has the form

$$\sin(A) - \widetilde{r}_m(A) = \sum_{i=m}^{\infty} c_{2i+1} A^{2i+1}.$$

Since this expansion contains only odd powers of $A$ we bound the series in terms of $\|A\|$ instead of $\theta(A)$ (cf. (4.5)):

$$\|\sin(A) - \widetilde{r}_m(A)\| \le \sum_{i=m}^{\infty} |c_{2i+1}| \beta^{2i}, \qquad \beta = \|A\|. \tag{4.13}$$

Define $\beta_m$ to be the largest value of $\beta$ such that the bound (4.13) does not exceed $u$. Using the same technique as for the cosine, we computed the values shown in Table 4.8. These values of $\beta_m$ can be compared with the values of $\theta_{2m}$ in Table 4.4. Although $\theta_{2m}$ is defined as the largest value of $\theta(A) = \|A^2\|^{1/2}$ such that the absolute error bound (4.12) for $\|\cos(A) - r_{2m}(A)\|$ does not exceed $u$, $\theta_{2m}$ can also (trivially) be regarded as the largest value of $\|A\|$ such that the bound (4.12), with $\theta$ interpreted as $\|A\|$, does not exceed $u$.

On comparing Table 4.8 with Table 4.4 we see that for $4 \le 2m \le 22$ we have $\beta_{2m} < \theta_{2m} < \beta_{2m+1}$. We could therefore scale so that $\|2^{-s}A\| \le \beta_{2m}$ and then use the $[2m/2m]$ Padé approximants to the sine and cosine, or scale so that $\|2^{-s}A\| \le \theta_{2m}$ and use the $[2m/2m]$ Padé approximant to the cosine

Table 4.9: Number of matrix multiplications $\widetilde{\pi}_{2m}$ to evaluate $p_{2m}(A)$, $q_{2m}(A)$, $\widetilde{p}_{2m+1}(A)$, and $\widetilde{q}_{2m+1}(A)$.

| $2m$ | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 | 22 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $\widetilde{\pi}_{2m}$ | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 10 | 11 | 11 |

and the $[2m+1/2m+1]$ Padé approximant to the sine. Since the diagonal Padé approximants to the sine have an odd numerator polynomial and an even denominator polynomial [39], and since we can write an odd polynomial in $A$ as $A$ times an even polynomial of degree one less, it is as cheap to evaluate $\widetilde{r}_{2m+1}$ and $r_{2m}$ as to evaluate $\widetilde{r}_{2m}$ and $r_{2m}$. Therefore we will scale so that $\|2^{-s}A\| \leq \theta_{2m}$ and then evaluate $r_{2m}$ for the cosine and $\widetilde{r}_{2m+1}$ for the sine. Evaluating $p_{2m}$, $q_{2m}$, $\widetilde{p}_{2m+1}$ and $\widetilde{q}_{2m+1}$ reduces to evaluating four even polynomials of degree $2m$ if we write $\widetilde{p}_{2m+1}$ as $A$ times an even polynomial of degree $2m$. This can be done by forming the powers $A^2$, $A^4$, ..., $A^{2m}$, at a total cost of $m+1$ multiplications. However, for $2m \geq 20$ it is more efficient to use the schemes of the form (4.7). We summarise the cost of evaluating $p_{2m}$, $q_{2m}$, $\widetilde{p}_{2m+1}$ and $\widetilde{q}_{2m+1}$ for $2m = 2\colon 2\colon 24$ in Table 4.9.

Now we consider the choice of degree, $2m$. Bounds analogous to those in Table 4.5 show that $\widetilde{q}_{2m+1}$ is well conditioned for $2m \leq 24$, and bounds for $\widetilde{p}_{2m+1}$ and $\widetilde{q}_{2m+1}$ analogous to those in Table 4.6 suggest restricting to $2m \leq 20$ (the same restriction that was made in Section 4.3 for the Padé approximants for the cosine). It is then clear from Table 4.9 that we need only consider $2m = 2, 4, 6, 8, 10, 12, 14, 16, 20$. Noting that dividing $A$ by 2 results in two extra multiplications in the double-angle phase and that increasing from one value of $2m$ to the next in our list of considered values increases the cost of evaluating the Padé approximants by one multiplication, we can determine the most efficient choice of $2m$ by a similar argument to that in the previous section. The result is that we should scale so that $\theta \leq \theta_{20}$, and scale further

according to exactly the same strategy as in Table 4.7, except for the fact that in the first line of the table "14" is added to the set of possible $d$ values.

The algorithm can be summarised as follows.

**Algorithm 4.5.1.** *Given a matrix $A \in \mathbb{C}^{n\times n}$ this algorithm approximates $C = \cos(A)$ and $S = \sin(A)$. It uses the constants $\theta_{2m}$ given in Table 4.4. The matrix $A$ can optionally be preprocessed using an obvious modification of Algorithm 4.1.2.*

```
1  for d = [2 4 6 8 12 14 16]
2      if ||A||_∞ ≤ θ_d
3          C = r_d(A), S = r̃_d(A)
4          quit
5      end
6  end
7  s = ceil(log_2(θ/θ_20)) % Find minimal integer s such that 2^{-s}θ ≤ θ_20.
8  Determine optimal d from modified Table 4.7 (with θ ← 2^{-s}θ)
   and increase s as necessary.
9  C = r_d(2^{-s}A), S = r̃_d(2^{-s}A)
10 for i = 1: s
11     S ← 2CS, C ← 2C^2 − I
12 end
```

**Cost**: $\big(\widetilde{\pi}_d + \mathrm{ceil}(\log_2(\|A\|_\infty/\theta_d))\big)M + D$, where $d$ is the degree of the Padé approximants used and $\theta_d$ and $\widetilde{\pi}_d$ are tabulated in Tables 4.4 and 4.9, respectively.

How much work does Algorithm 4.5.1 save compared with separate computation of $\cos(A)$ and $\sin(A) = \cos(A - \frac{\pi}{2}I)$ by Algorithm 4.3.1? The answer is roughly $2\pi_d - \widetilde{\pi}_d$ matrix multiplies, which rises from 1 when $d = 4$ to 4 when $d = 20$; the overall saving is therefore up to about 27%.

We tested Algorithm 4.5.1 on the same set of test matrices as in Section 4.4. Figure 4.5 compares the relative errors for the computed sine and cosine with the corresponding errors from `funm`, invoked as `funm(A,@sin)` and

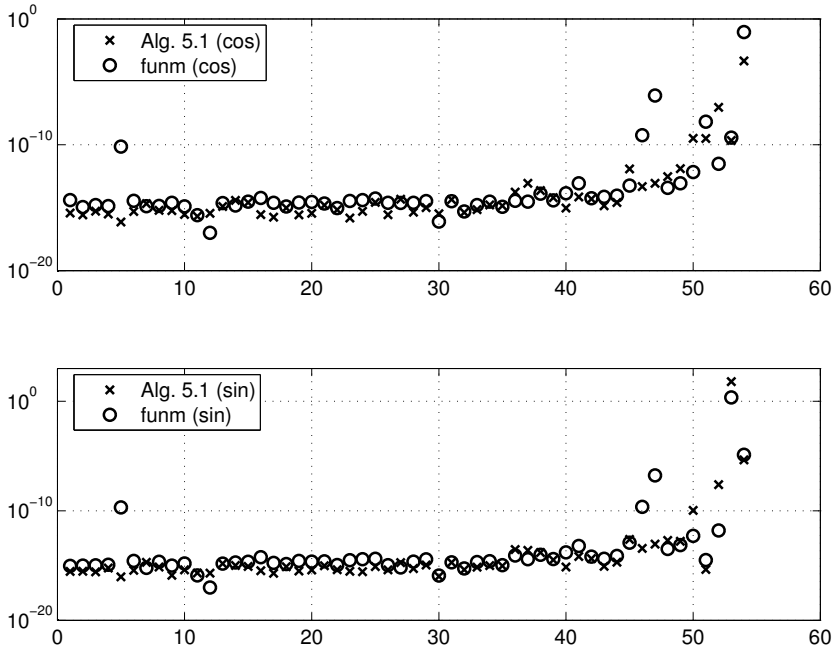


Figure 4.5: Errors for Algorithm 4.5.1 without preprocessing and `funm`.

`funm(A,@cos)`. Note that the cost of the latter two computations can be reduced by using the same Schur decomposition in both cases. Algorithm 4.5.1 provides similar or better accuracy to `funm` on this test set. Its cost varies from 9 matrix multiplies and solves to 54, with an average of 16, so the algorithm can require significantly fewer flops than are needed for a single Schur decomposition.

## 4.6 Conclusion

We have improved the algorithm of Higham and Smith [34] in two respects: by employing variable degree Padé approximants, with the degree chosen to minimise the computational cost, and by employing truncation error bounds expressed in terms of $\|A^2\|^{1/2}$ in place of $\|A\|$. Our two improved algorithms, Algorithms 4.2.1 and 4.3.1, both out-perform the Higham and Smith algorithm

in accuracy and cost. Of the two, Algorithm 4.3.1, based on an absolute error bound for the Padé approximant, emerges as the clear winner. By its design, it allows larger degree Padé approximants to be evaluated at matrices of significantly larger norm, but in so doing it does not sacrifice accuracy, as we have shown by analysis (see (4.11)) and experiment. Analogously to our experience with the matrix exponential [32], designing our algorithms to achieve low cost brings an added benefit of better accuracy through the need for fewer double-angle steps.

We have also shown how, using the Padé double-angle approach, $\cos(A)$ and $\sin(A)$ can be evaluated together at lower cost than if they are evaluated separately.

The design of the algorithms involved making compromises between maximising efficiency and minimising the effects of rounding errors. Compared with the Schur–Parlett method applied to the sine and cosine the algorithms require fewer flops unless $\|A^2\|^{1/2}$ is large, and on our test set they are generally more accurate; this provides confidence that the compromises have been well chosen.

# Chapter 5

# Summary

A collection of algorithms has been presented for applying $J$-orthogonal and $(J_1, J_2)$-orthogonal transformations. This includes a variety of methods for constructing and applying hyperbolic rotations, fast hyperbolic rotations, unified rotations and hyperbolic Householder transformations. We have provided an extensive rounding error analysis to identify which of these methods are stable and have given numerical experiments to show which are not. We also considered the hyperbolic QR factorization, including new theorems for its existence. However, there are several issues that still warrant further investigation. We have been unable to find a stable method for constructing a hyperbolic rotation that zeros the second component of a complex vector. The problem is that we are unable to compute $x_1^2 + x_2^2 - x_3^2 - x_4^2$, for $x_i \in \mathbb{R}$, accurately. If this problem could be solved then it may be possible to extend the method to compute $x^T J x$, where $x \in \mathbb{R}^n$ and $J = \text{diag}(\pm 1)$. This would then provide a stable method of applying hyperbolic Householder transformations, for which it is currently unclear if this is possible. We have shown that nonoverlapping hyperbolic transformations can be applied in a stable way, but we are uncertain of the stability properties for overlapping hyperbolic rotations, and feel that further investigation would be of interest.

We have presented two new algorithms for computing the 1-norm condition number of a tridiagonal matrix in $O(n)$ operations, which avoid underflow and overflow, and are simpler than existing algorithms. A rounding error analysis of the first algorithm shows it to be stable, while numerical experiments suggest that the second algorithm is also stable and competitive in speed with existing algorithms. We have been unable to prove the second algorithm to be stable but feel that it may be possible with further investigation. It has been proposed that our codes for these algorithms should be included in a future release of LAPACK. We then considered diagonal-plus-semiseparable matrices and showed how it is possible to compute the 1-norm condition number of such matrices in $O(n)$ operations. There are many other structured matrices $A \in \mathbb{R}^{n\times n}$ such that the linear system $Ax = b$ can be solved in $O(n)$ operations but for which it is still an open problem to compute the condition number in $O(n)$ operations. For example, it would be useful to be able to compute the condition number of a pentadiagonal matrix or a general banded matrix in $O(n)$ operations.

Two new algorithms for computing the matrix cosine have been presented. Numerical experiments and theory show improved accuracy and efficiency over existing methods. We also extended the ideas developed for the matrix cosine to present an algorithm that simultaneously computes the matrix cosine and sine, and does so more efficiently than if they were computed separately. The ideas used in the design of the algorithms could be extended to various other matrix functions. Algorithms for computing $\cosh(A)$, or $\cosh(A)$ and $\sinh(A)$ simultaneously, are easily obtained by adapting the algorithms for their trigonometric counterparts. However, it is not possible to compute $\sinh(A)$ by making use of $\cosh(A)$ unlike $\sin(A) = \cos(A - \frac{\pi}{2}I)$. Therefore, it may be of interest to investigate an algorithm that scales the matrix by powers of 3 and makes use of the triple angle formula $\sinh(A) = 3\sinh(A) + 4\sinh^3(A)$.

It may also be interesting to consider an algorithm that computes $\tan(A)$ by making use of scaling, a Padé approximation and the double angle formula $\tan(2A) = 2\tan(A)(I - \tan^2(A))^{-1}$.

# Appendix A

# Hyperbolic Transformations Toolbox for MATLAB

The algorithms described in Chapter 2 have been implemented in MATLAB and are presented here. A summary of the M-files is given by the following tables.

| **Transformations** | |
|---|---|
| `hrotate` | Constructs a hyperbolic rotation using formulae H3 or H4. |
| `happly` | Applies a hyperbolic rotation directly or in mixed form. |
| `happly_od` | Applies a hyperbolic rotation by the OD procedure. |
| `urotate` | Constructs a unified rotation equivalent to formulae H3 or H4. |
| `uapply` | Applies a unified rotation directly or in mixed form. |
| `uapply_od` | Applies a unified rotation using the OD procedure. |
| `grotate_fast` | Constructs a fast Givens rotation. |
| `gapply_fast` | Applies a fast Givens rotation. |
| `hrotate_fast` | Constructs a fast hyperbolic rotation |
| `happly_fast` | Applies a fast hyperbolic rotation. |
| `hhouse` | Computes hyperbolic Householder vector. |
| `uhouse` | Computes $v$ and $k$ that define a unified Householder matrix. |

| **Hyperbolic QR Factorization** | |
|---|---|
| `hqr` | Hyperbolic QR factorization using nonoverlapping hyperbolic rotations applied directly or in mixed form. |
| `hqr_od` | Hyperbolic QR factorization using nonoverlapping hyperbolic rotations applied by the OD procedure. |

| Applications | |
|---|---|
| `choldown` | Solves the Cholesky downdating problem by hyperbolic QR. |
| `ils` | Solves the indefinite least squares problem by hyperbolic QR. |

## A.1 Transformations

```
function [c,s] = hrotate(x,rep)
%HROTATE    Constructs a hyperbolic rotation.
%           [c,s] = HROTATE(x,rep) calculates scalars c and s
%           that define a hyperbolic rotation H = [CONJ(c)
%           -CONJ(s); -s c] with ABS(c)^2-ABS(s)^2 = 1 such
%           that H*x, (x a 2*1 real or complex vector) has a
%           zero second element.  The representation of the
%           hyperbolic rotation depends on rep.
%           If rep = 1 (default) : x1^2-x2^2 computed by
%                                  (x1+x2)(x1-x2),
%              rep = 2 : x1^2-x2^2 computed by 2e-e^2, where
%                                e = (abs(x1)-abs(x2))/abs(x1).

if nargin < 2
    rep = 1;
end

if abs(x(1)) <= abs(x(2))
    error('Hyperbolic rotation only exists if |x(1)|>|x(2)|')
end

if x(2) == 0
    c = 1; s = 0;
elseif rep ~= 2
    t = sqrt((abs(x(1))+abs(x(2)))*(abs(x(1))-abs(x(2))));
    c = x(1)/t; s = x(2)/t;
else
    e = (abs(x(1))-abs(x(2)))/abs(x(1));
    c = sign(x(1))/sqrt(2*e-e^2);
    s = (x(2)/x(1))*c;
end



function X = happly(c,s,X,app)
%HAPPLY  Applies a hyperbolic rotation.
```

```
%        Y = HAPPLY(c,s,X,app) applies the hyperbolic rotation
%        matrix H = [CONJ(c),-CONJ(s);-s,c] to the 2*n complex
%        matrix X in either direct form or mixed form,
%        producing Y=H*X.  If app = 1 then the direct method
%        is used and if app = 2 then the mixed method is
%        used (default).

if nargin < 3
    error('At least 3 input arguments must be supplied.')
elseif nargin == 3
    app = 2;
end

n = size(X,2);
if app ~= 2      % Direct method.
    temp = X(1,:);
    X(1,:) = conj(c)*X(1,:)-conj(s)*X(2,:);
    X(2,:) = -s*temp+c*X(2,:);
else             % Mixed method
    X(1,:) = conj(c)*X(1,:)-conj(s)*X(2,:);
    X(2,:) = (-s*X(1,:)+X(2,:))/conj(c);
end
```

```
function X = happly_od(x1,x2,X)
%HAPPLY_OD  Applies a hyperbolic rotation by the OD-procedure.
%           Y = HAPPLY_OD(x1,x2,X) applies the hyperbolic
%           rotation, H, that zeros the second component of a
%           real vector [x1;x2] to the real matrix X so that
%           Y = H*X.

if nargin < 3
    error('At least 3 input arguments must be supplied.')
end
if abs(x1) <= abs(x2)
    error('Hyperbolic rotation only exists if |x1|>|x2|')
end
n = size(X,2);
d = sqrt((x1+x2)/(x1-x2));
X = [1 -1; 1 1]*X;
X = [d/2 0; 0 1/(2*d)]*X;
X = [1 1;-1 1]*X;
```

```
function [c,s,J2] = urotate(x,J1,rep)
%UROTATE    Constructs a unified rotation.
%           [c,s,J2] = UROTATE(x,J1,rep) computes scalars c and s
%           that define a unified rotation
%           H = [CONJ(c),J1(1,1)*J1(2,2)*CONJ(s);-s,c] such that
%           H*x, (x is a real or complex 2*1 vector) has a zero
%           second element.  If J1 = +/-EYE(2) then a Givens
%           rotation is defined, otherwise a Type 1 or Type 2
%           hyperbolic rotation is defined.  H is
%           (J1,J2)-orthogonal, where J2 = H'*J1*H.
%           If rep = 1 (default) : x1^2-x2^2 computed by
%                                        (x1+x2)(x1-x2)
%              rep = 2 : x1^2-x2^2 computed by 2e-e^2, where
%                        e = (abs(x1)-abs(x2))/abs(x1).

if nargin < 2
    error('At least 2 input arguments must be suplied.')
elseif nargin == 2
    rep = 1;
end
m = J1(1,1)*J1(2,2);
J2 = J1;
if x(2) == 0
    s = 0; c = 1;
    return
end

if J1(1,1)*abs(x(1)) == -J1(2,2)*abs(x(2))
    error('Hyperbolic rotation not defined for |x(1)| = |x(2)|.')
end
if abs(x(1)) > abs(x(2))
    if m == 1            % Givens rotation.
        t = x(2)/x(1);
        d = 1+abs(t)^2;
        c = sign(x(1))/sqrt(abs(d)); s = t*c;
    else                 % Hyperbolic type 1 rotation.
        if rep ~= 2
            t = sqrt((abs(x(1))+abs(x(2)))*(abs(x(1))-abs(x(2))));
            c = x(1)/t; s = x(2)/t;
        else
            e = (abs(x(1))-abs(x(2)))/abs(x(1));
            d = sqrt(2*e-e^2);
            c = sign(x(1))/d;
            s = (x(2)/x(1))/d;
```

```
        end
    end
else
    if m == 1            % Givens rotation.
        t = x(1)/x(2);
        d = 1+abs(t)^2;
        s = sign(x(2))/sqrt(d); c = t*s;
    else                 % Hyperbolic type 2 rotation.
        J2 = -J2;
        if rep ~= 2
            t = sqrt((abs(x(1))+abs(x(2)))*(abs(x(2))-abs(x(1))));
            c = x(1)/t; s = x(2)/t;
        else
            e = (abs(x(2))-abs(x(1)))/abs(x(2));
            s = sign(x(2))/sqrt(2*e-e^2);
            c = (x(1)/x(2))*s;
        end
    end
end
```

```
function X = uapply(c,s,X,J1,J2,app)
%UAPPLY     Applies a unified rotation.
%           Y = UAPPLY(c,s,X,J1,J2,app) applies the unified
%           rotation H = [CONJ(c),J1(1,1)*J1(2,2)*CONJ(s);-s,c],
%           to the 2*n complex matrix X in either direct form or
%           mixed form, producing Y = H*X.  J1 and J2 are
%           signature matrices such that H is (J1,J2)-orthogonal.
%           If app = 1 then the direct method used and if app = 2
%           the mixed method is used (default).

if nargin < 5
    error('At least 5 input arguments must be supplied.')
elseif nargin == 5
    app = 2;
end

t = J1(1,1)*J1(2,2);
temp = X(1,:);
X(1,:) = conj(c)*X(1,:)+t*conj(s)*X(2,:);
if app ~= 2                    % Direct method.
    X(2,:) = -s*temp+c*X(2,:);
else                           % Mixed method.
```

```
    if t == 1                    % Givens rotation.
        X(2,:) = -s*temp+c*X(2,:);
    elseif J1(1,1) == J2(1,1);   % Hyperbolic type 1 rotation.
        X(2,:) = (-s*X(1,:)+X(2,:))/conj(c);
    else                         % Hyperbolic type 2 rotation.
        X(2,:) = (-c*X(1,:)-temp)/conj(s);
    end
end



function [X,J2] = uapply_od(x1,x2,X,J1)
%UAPPLY_OD  Applies a unified rotation using the OD procedure.
%           [Y,J2] = UAPPLY_OD(x1,x2,X,J1) applies a unified
%           rotation, H, that zeros the second component of the
%           real vector [x1;x2], to the 2*n matrix X, producing
%           Y = H*X.  Given the 2 by 2 signature matrix J1, the
%           unified rotation is (J1,J2)-orthogonal, where J2 is a
%           signature matrix.  The Type 1 and Type 2 hyperbolic
%           rotations are applied using the OD procedure.

if nargin < 4
    error('At least 4 input arguments must be supplied.')
end

if J1(1,1)*abs(x1) == -J1(2,2)*abs(x2)
    error('Hyperbolic rotation not defined for |x(1)| = |x(2)|.')
end

s = J1(1,1)*J1(2,2);
if s == 1
    if abs(x1)>abs(x2)            % Givens rotation
        t = x2/x1;
        d = 1+abs(t)^2;
        c = sign(x1)/sqrt(abs(d)); s = t*c;
    else
        t = x1/x2;
        d = 1+abs(t)^2;
        s = sign(x(2))/sqrt(d); c = t*s;
    end
    temp = X(1,:);
    X(1,:) = conj(c)*X(1,:)+conj(s)*X(2,:);
    X(2,:) = -s*temp+c*X(2,:);
    J2 = J1;
```

```
else
    if abs(x1) > abs(x2)
        m = 1;                      % Type 1 hyperbolic rotation
    else
        m = -1;                     % Type 2 hyperbolic rotation
    end
    d = sqrt((x1+x2)/m*(x1-x2));
    J2 = m*J1;
    X = [1 -1; 1 1]*X;
    X = [d/2 0; 0 1/(2*d)]*X;
    X = [m 1;-m 1]*X;
end



function [alpha,beta,k,type] = grotate_fast(x,d)
%GROTATE_FAST     Computes a fast Givens rotation.
%       [alpha,beta,k,type] = GROTATE_FAST(x,d) computes the
%       scalars alpha, beta and vector k which define the fast
%       Givens rotation.  The vector d is the original diagonal
%       factor and k is the updated diagonal factor.  The fast
%       rotation zeros the second component of 2*1 vector
%       y = DIAG(d)*x (x must be real).  The variable type
%       specifies the form of fast Givens rotation used and
%       determines how it is applied.

x1 = x(1); x2 = x(2);
d1 = d(1); d2 = d(2);
d1x1 = d1*x1; d2x2 = d2*x2;
k = zeros(2,1);

if x2 == 0
    type = 1;
    beta = 0; alpha = 0;
    return
end

if abs(d1x1) >= abs(d2x2)
    c = 1/sqrt(1+(d2x2/d1x1)^2);
    s = (d2x2*c)/d1x1;
else
    s = 1/sqrt(1+(d1x1/d2x2)^2);
    c = (d1x1*s)/d2x2;
end
```

```
if c >= 0.5
    alpha = -x2/x1;
    beta = -alpha*(d2/d1)^2;
    k(1) = c*d1; k(2) = c*d2;
    type = 2;
else
    alpha = x1/x2;
    beta = alpha*(d1/d2)^2;
    k(1) = s*d2; k(2) = s*s1;
    type = 3;
end
```

```
function A = gapply_fast(A,alpha,beta,type)
%GAPPLY_FAST  Applies a fast Givens rotation.
%             Y = GAPPLY_FAST(A,alpha,beta,type) applies the
%             fast Givens rotation, defined by alpha and beta,
%             to the 2*n matrix A, producing Y.

if nargin < 4
    error('At least 2 input arguments must be supplied')
end
if type == 2
    tmp = A(1,:);
    A(1,:) = A(1,:)+beta*A(2,:);
    A(2,:) = alpha*tmp+A(2,:);
elseif type == 3
    tmp = A(1,:);
    A(1,:) = beta*A(1,:)+A(2,:);
    A(2,:) = -tmp+alpha*A(2,:);
end
```

```
function [alpha,beta,k,type] = hrotate_fast(x,d)
%HROTATE_FAST    Computes a fast hyperbolic rotation.
%       [alpha,beta,k,type] = HROTATE_FAST(x,d) computes the
%       scalars alpha, beta and vector k which define the
%       fast hyperbolic rotation.  The vector d is the original
%       diagonal factor and k is the updated diagonal factor.
%       The fast hyperbolic rotation zeros the second component
```

```
%       of the 2*1 vector y = DIAG(d)*x (x must be real), and the
%       representation of fast rotation is chosen to reduce the
%       risk of underflow and overflow.  The variable type
%       specifies the form of fast hyperbolic rotation used
%       and determines how it is applied.

x1 = x(1); x2 = x(2);
d1 = d(1); d2 = d(2);
d1x1 = d1*x1; d2x2 = d2*x2;

if abs(d1x1) <= abs(d2x2)
    error('Fast hyperbolic rotation not defined')
end

e = (abs(d1x1)-abs(d2x2))/abs(d1x1);
c = 1/sqrt(2*e-e^2);
s = (d2x2/d1x1)*c;


if x2 == 0
    type=1;
    beta=0; alpha=0;
elseif abs(d1) >= abs(d2)
    alpha = (d1*c*s)/(d2);
    beta = (d2*s)/(d1*c);
    k(1) = c*d1; k(2) = d2/c;
    type = 2;
else
    alpha = (s*d1)/(c*d2);
    beta = c*s*d2/d1;
    k(1) = d1/c; k(2) = d2*c;
    type = 3;
end




function A = happly_fast(A,alpha,beta,type)
%HAPPLY_FAST Applies a fast hyperbolic rotation.
%          Y = HAPPLY_FAST(A,alpha,beta,type) applies the fast
%          hyperbolic rotation, defined by alpha and beta, to the
%          2*n matrix A, producing Y.

if nargin < 4
    error('At least 2 input arguments must be supplied')
```

```
end
if type == 2
    A(1,:) = A(1,:)-beta*A(2,:);
    A(2,:) = A(2,:)-alpha*A(1,:);
elseif type == 3
    A(2,:) = A(2,:)-alpha*A(1,:);
    A(1,:) = A(1,:)-beta*A(2,:);
end
```

```
function [v,beta] = hhouse(x,J,k)
%HHOUSE     Computes hyperbolic Householder vector.
%           [v,beta] = HHOUSE(x,J,k) computes the hyperbolic
%           Householder vector v that defines the hyperbolic
%           transformation H = J-(2*v*v')/(v'*J*v) such that H*x
%           zeros all but the kth component of x which is equal
%           to beta. H is J-orthogonal.

if nargin < 3, k = 1; end

t = x'*(diag(J).*x);

if t == 0
    error('Hyperbolic Householder transformation does not exist.')
end

if sign(t) == sign(J(k,k))
    beta = -J(k,k)*(x(k)/abs(x(k)))*sqrt(abs(t));
    v = J*x;
    v(k) = v(k)-beta;
else
    error('Hyperbolic Householder transformation does not exist.')
end
```

```
function [v,k,beta] = uhouse(x,J,opt)
%UHOUSE  Computes variables defining a unified Householder matrix.
%        [v,k] = UHOUSE(x,J,opt) computes v and k that define
%        the unified Householder matrix H = P(J-(2vv')/(v'Jv))
%        such that all but the first element of y = H*x are
%        zero and the first element is beta.  P is a permutation
```

```
%        matrix which swaps the first and pth rows.

if nargin < 3, opt = 0; end

t = x'*(diag(J).*x);
if t == 0
    error('Hyperbolic Householder transformation does not exist.')
end

index = find(diag(J) == sign(t));
if opt == 0
    k = index(1);
else
    [val,k] = max(abs(x(index)));
    k = index(k);
end
beta = -J(k,k)*(x(k)/abs(x(k)))*sqrt(abs(t));

v = J*x;
v(k) = v(k)-beta;
```

## A.2 Hyperbolic QR Factorization

```
function [Q,R,B] = hqr(A,p,rep,app,B)
%HQR    Hyperbolic QR factorization using hyperbolic rotations.
%       [Q,R,C] = HQR(A,p,rep,app,B) computes an upper triangular
%       R and J-orthogonal Q, where Q*A = R and
%       J = DIAG(I_p,I_(m-p)). If B (an m*s matrix) is given then
%       C is returned as C = Q*B.  Two different representations
%       of hyperbolic rotations and two methods of applying these
%       can be chosen.
%       rep: representation of hyperbolic rotation used.
%              (default=1)
%              rep = 1 : x1^2-x2^2 computed by (x1+x2)(x1-x2)
%              rep = 2 : x1^2-x2^2 computed by 2e-e^2, where
%                            e = (abs(x1)-abs(x2))/abs(x1)
%       app: method of applying hyperbolic rotation used.
%              (default=1)
%              app = 1 : direct method used
%              app = 2 : mixed metrhod used
%       R = hqr(A,p,rep,app,B) only outputs upper triangular R.

[m,n] = size(A);
if nargin < 2
```

```
    error('At least 2 input arguments must be supplied')
elseif nargin == 2
    rep = 1; app = 1;
elseif nargin == 3
    app = 1;
end

Q = eye(m);
if isequal(A(1:p,:),triu(A(1:p,:))) == 0
    [Q(1:p,1:p),A(1:p,:)] = qr(A(1:p,:));
    Q = Q';
    if nargin > 4
        B(1:p,:) = Q(1:p,1:p)*B(1:p,:);
    end
end

for j = 1:min([m-1,n,p])          % Householder transformation.
    [v,beta,s] = gallery('house',A(p+1:m,j));
    A(p+1:m,j) = [s; zeros(m-p-1,1)];
    tmp = v'*A(p+1:m,j+1:n);
    A(p+1:m,j+1:n) = A(p+1:m,j+1:n)-beta*v*tmp;
    tmp = v'*Q(p+1:m,:);
    Q(p+1:m,:) = Q(p+1:m,:)-beta*v*tmp;
    if nargin > 4
        tmp = v'*B(p+1:m,:);
        B(p+1:m,:) = B(p+1:m,:)-beta*v*tmp;
    end
    [c,s] = hrotate(A([j,p+1],j),rep);   % Hyperbolic rotation.
    A([j p+1],j:n) = happly(c,s,A([j,p+1],j:n),app);
    Q([j,p+1],:) = happly(c,s,Q([j,p+1],:),app);
    if nargin > 4
        B([j,p+1],:) = happly(c,s,B([j,p+1],:),app);
    end
    A(j+1:m,j) = 0;
end

if p < n
    [Q1,A(p+1:end,p+1:end)] = qr(A(p+1:end,p+1:end));
    Q(p+1:end,:) = Q1'*Q(p+1:end,:);
    if nargin > 4
        B(p+1:end,:) = Q1'*B(p+1:end,:);
    end
end
if nargout == 1
```

```
    Q = A;
else
    R = A;
end



function [Q,R,B] = hqr_od(A,p,B)
%HQR_OD Hyperbolic QR factorization using hyperbolic rotations.
%       [Q,R,C] = HQR_OD(A,p,rep,app,B) computes an upper
%       triangular R and J-orthogonal Q, where Q*A = R and
%       J = DIAG(I_p,I_(m-p)).  The matrix A must be real.
%       If B (an m*s matrix) is given then C is returned as
%       C = Q*B.  Hyperbolic rotations are applied using the
%       OD procedure.  Also R = hqr_od (A,p,rep,app,B) only
%       outputs upper triangular R.

[m,n] = size(A);
if nargin < 2
    error('At least 2 input arguments must be supplied')
end

Q = eye(m);
if isequal(A(1:p,:),triu(A(1:p,:))) == 0
    [Q(1:p,1:p),A(1:p,:)] = qr(A(1:p,:));
    Q = Q';
    if nargin > 2
        B(1:p,:) = Q(1:p,1:p)*B(1:p,:);
    end
end

for j = 1:min([m-1,n,p])       % Householder transformation.
    [v,beta,s] = gallery('house',A(p+1:m,j));
    A(p+1:m,j) = [s; zeros(m-p-1,1)];
    tmp = v'*A(p+1:m,j+1:n);
    A(p+1:m,j+1:n) = A(p+1:m,j+1:n)-beta*v*tmp;
    tmp = v'*Q(p+1:m,:);
    Q(p+1:m,:) = Q(p+1:m,:)-beta*v*tmp;
    if nargin > 2
        tmp = v'*B(p+1:m,:);
        B(p+1:m,:) = B(p+1:m,:)-beta*v*tmp;
    end
    x1 = A(j,j); x2 = A(p+1,j);      % Hyperbolic rotation.
    A([j p+1],j:n) = happly_od(x1,x2,A([j,p+1],j:n));
```

```
    Q([j,p+1],:) = happly_od(x1,x2,Q([j,p+1],:));
    if nargin > 2
        B([j,p+1],:) = happly_od(x1,x2,B([j,p+1],:));
    end
    A(j+1:m,j) = 0;
end

if p < n
    [Q1,A(p+1:end,p+1:end)] = qr(A(p+1:end,p+1:end));
    Q(p+1:end,:) = Q1'*Q(p+1:end,:);
    if nargin > 2
        B(p+1:end,:) = Q1'*B(p+1:end,:);
    end
end
if nargout == 1
    Q = A;
else
    R = A;
end
```

## A.3 Applications

```
function R = choldown(A,B,rep,app)
%CHOLDOWN   Cholesky downdating problem using hyperbolic QR.
%           R = CHOLDOWN(A,B,rep,app) computes the Cholesky
%           factor, R, of C = A'*A-B'*B by computing the
%           hyperbolic QR factorization of [A;B] so that
%           R'*R = C.  A is a p*n matrix and B is a q*n matrix.
%           Two different representations of hyperbolic rotations
%           and two methods of applying these can be chosen.
%           rep: representation of hyperbolic rotation.
%           (default=1)
%                  rep = 1 : x1^2-x2^2 computed by (x1+x2)(x1-x2)
%                  rep = 2 : x1^2-x2^2 computed by 2e-e^2, where
%                                  e = (abs(x1)-abs(x2))/abs(x1)
%           app: method of applying hyperbolic rotation.
%           (default=2)
%                  app = 1 : direct method used
%                  app = 2 : mixed method used

if nargin < 2
```

```
    error('Must supply variables A and B')
elseif nargin == 2
    rep = 1; app = 2;
elseif nargin == 3
    app = 2;
end

[p,n] = size(A);
[q,n2] = size(B);
if n ~= n2
    error('A and B must have the same number of columns')
end

R = hqr([A;B],p);



function x = ils(A,b,p,rep,app)
%ILS    Indefinite least squares problem by hyperbolic QR.
%       ILS(A,b,p) solves the indefinite least squares problem
%       min(b-Ax)'J(b-Ax) over x, where
%       J = diag(eye(p),-eye(m-p)).  Two different
%       representations of hyperbolic rotations and two methods
%       of applying these can be chosen.
%       rep: representation of hyperbolic rotation used.
%       (default=1)
%              rep = 1 : x1^2-x2^2 computed by (x1+x2)(x1-x2)
%              rep = 2 : x1^2-x2^2 computed by 2e-e^2, where
%                              e = (abs(x1)-abs(x2))/abs(x1)
%       app: method of applying hyperbolic rotation used.
%       (default=2)
%              app = 1 : direct method used
%              app = 2 : mixed method used

if nargin < 3
    error('At least 3 input arguments must be supplied.')
elseif nargin == 3
    rep = 1; app = 2;
elseif nargin == 4
    app = 2;
end

[m,n] = size(A);
if m < n
```

```
    error('m*n matrix A must have m >= n')
end
[Q,R,b] = hqr(A,p,b,rep,app);     % Hyperbolic QR factorization.
x = R(1:n,:)\b(1:n);
```

# Appendix B

# LAPACK Style Codes for the Condition Number of a Tridiagonal Matrix

Algorithms 3.2.2 and 3.2.3 have been implemented in the style of LAPACK using Fortran 77. The codes for both double precision and complex double precision tridiagonal matrices are given here.

## B.1 Double Precision Codes

```
      SUBROUTINE DGTCN1(N,b,a,c,rcond,work,info)
*
*     .. Scalar Arguments ..
      INTEGER            N, info
      DOUBLE PRECISION   rcond
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   a( * ), b( * ), c( * ), WORK( * )
*     ..
*
*     Purpose
*     =======
*
*     DGTCN1 computes the reciprical of the 1-norm condition number
*     of a real tridiagonal matrix A using Algorithm 3.2.2.  At
*     most 61N operations are required.
*
*     Arguments
*     =========
```

```
*
*     N       (input) INTEGER
*             The order of the matrix A.  N >= 0.
*
*     a       (input) DOUBLE PRECISION array, dimension (N)
*             The n diagonal elements of the tridiagonal matrix A.
*
*     b       (input) DOUBLE PRECISION array, dimension (N-1)
*             The (n-1) subdiagonal elements of the tridiagonal
*             matrix A.
*
*     c       (input) DOUBLE PRECISION array, dimension (N-1)
*             The (n-1) superdiagonal elements of the tridiagonal
*             matrix A.
*
*     rcond   (output) DOUBLE PRECISION
*             The reciprical of the 1-norm condition number of A.
*
*     WORK    (workspace) DOUBLE PRECISION array, dimension (6*N)
*
*     INFO    (output) INTEGER
*             INFO = 0 if the matrix is nonsingular.
*             INFO = -i, the i-th argument had an illegal value.
*
*
*  =====================================================================
*
*     .. Parameters ..
      DOUBLE PRECISION   ONE, ZERO
      PARAMETER          ( ONE = 1.0D+0, ZERO = 0.0D+0 )
*     ..
*     .. Local Scalars ..
      DOUBLE PRECISION   temp,g,ci,tmp2,inorm,tnorm
      INTEGER            i,n2,n3,n4,n5
*
*     .. Intrinsics ..
      INTRINSIC          ABS
*
*     .. External Functions ..
      DOUBLE PRECISION   DLANGT
      EXTERNAL           DLANGT
*
*     Test the input arguments.
*
```

```
      info = 0
      if( N.LT.0 ) then
         INFO = -2
      end if
      if( INFO.NE.0 ) then
         call XERBLA( 'algorithm1', -INFO )
         return
      end if
*
*     Computes the 1-norm of A
*
      tnorm = DLANGT('1',n,b,a,c)
      inorm = 0
*
*     Quick return if possible
*
       if (n.eq.zero) then
         rcond = one
         return
      end if
      IF( tnorm.EQ.ZERO ) THEN
         rcond = zero
         RETURN
      end if
      IF( N.eq.1) THEN
         RCOND = one
         RETURN
      END IF
*
*     Partition Workspace
*
      n2 = 2*n
      n3 = 3*n
      n4 = 4*n
      n5 = 5*n
*
*     QR factorization
*
      temp = a(1)
      g = c(1)
      do 10 i = 1, n-1
         call DLARTG(temp,b(i),ci,tmp2,work(i))
         if (work(i) == 0) then
            rcond = 0
```

```
            return
         endif
         work(n+i) = -tmp2*(ci*g+tmp2*a(i+1))
         temp = -tmp2*g+ci*a(i+1)
         work(n3+i) = ci
         if (i < n-1) then
            work(n2+i) = tmp2**2*c(i+1)
            g = ci*c(i+1)
         endif
         if (i > 1) then
            work(n2+i-1) = tmp2*work(n2+i-1)
         endif
         work(n4+i) = tmp2
 10   continue
      work(n) = temp
      if (work(n) == 0) then
         rcond = 0
         return
      endif
      work(n3+n) = 1.0
*
*     Solve linear system
*
      work(n) = work(n3+n)/work(n)
      work(n-1) = (work(n3+n-1)-work(n)*work(n+n-1))/work(n-1)
      do 20 i = n-2,1,-1
         work(i) = (work(n3+i)-work(i+1)*work(n+i)-work(i+2)*
     $        work(n2+i))/work(i)
 20   continue
*
*     Compute 1-norm of columns of strictly lower triangular part
*
      work(n5+n) = 0
      work(n5+n-1) = ABS(work(n4+n-1)*work(n))
      do 30 i = n-1,2,-1
         work(n5+i-1) = (work(n5+i)+ABS(work(i)))*ABS(work(n4+i-1))
 30   continue
      do 35 i = 2,n
         work(n5+i) = work(n5+i)*ABS(work(n3+i-1))
 35   continue
*
*     QL factorization
*
      temp = a(n)
```

```
      g = b(n-1)
      do 40 i = n,2,-1
         call DLARTG(temp,c(i-1),ci,tmp2,work(i))
         if (work(i) == 0) then
            rcond = 0
            return
         endif
         work(n+i) = -tmp2*(ci*g+tmp2*a(i-1))
         temp = -tmp2*g+ci*a(i-1)
         work(n3+i) = ci
         if (i > 2) then
            work(n2+i) = tmp2**2*b(i-2)
            g = ci*b(i-2)
         end if
         if (i < n) then
            work(n2+i+1) = tmp2*work(n2+i+1)
         end if
         work(n4+i) = tmp2
 40   continue
      work(1) = temp
      if (work(1) == 0) then
         rcond = 0
         return
      endif
      work(n3+1) = 1.0
*
*     Solve linear system
*
      work(1) = work(n3+1)/work(1)
      work(2) = (work(n3+2)-work(1)*work(n+2))/work(2)
      do 50 i = 3,n
         work(i) = (work(n3+i)-work(i-1)*work(n+i)-work(i-2)*
     $        work(n2+i))/work(i)
 50   continue
*
*     Compute 1-norm of columns of upper triangular part
*
      work(n2+1) = ABS(work(1))
      do 60 i = 1,n-1
         work(n2+i+1) = work(n2+i)*ABS(work(n4+i+1))+ABS(work(i+1))
 60   continue
*
*     Add parts together to get the condition number
*
```

```
      inorm = work(n5+n)+work(n2+n)
      do 70 i = 1, n-1
         temp = ABS(work(n3+i+1))*(work(n2+i)) +
     $        work(n5+i)
         if(temp > inorm) then
            inorm = temp
         endif
 70   continue
      rcond = (1.0/inorm)/tnorm
*
*     End of DGTCN1
*
      end




      subroutine DGTCN2(N,b,a,c,rcond,work,info)
*
*     .. Scalar Arguments ..
      INTEGER            N,info
      DOUBLE PRECISION   rcond
*     ..
*     .. Array Arguments ..
      DOUBLE PRECISION   a( * ), b( * ), c( * ), WORK( * )
*     ..
*
*     Purpose
*     =======
*
*     DGTCN2 computes the reciprical of the 1-norm condition number
*     of a real tridiagonal matrix A using Algorithm 3.2.3.  At
*     most 35N operations are required.
*
*     Arguments
*     =========
*
*     N       (input) INTEGER
*             The order of the matrix A.  N >= 0.
*
*     a       (input) DOUBLE PRECISION array, dimension (N)
*             The n diagonal elements of the tridiagonal matrix A.
*
```

```
*     b       (input) DOUBLE PRECISION array, dimension (N-1)
*             The (n-1) subdiagonal elements of the tridiagonal
*             matrix A.
*
*     c       (input) DOUBLE PRECISION array, dimension (N-1)
*             The (n-1) superdiagonal elements of the tridiagonal
*             matrix A.
*
*     rcond   (output) DOUBLE PRECISION
*             The reciprical of the 1-norm condition number of A.
*
*     WORK    (workspace) DOUBLE PRECISION array, dimension (5*N)
*
*     INFO    (output) INTEGER
*             INFO = 0 if the matrix is nonsingular.
*             INFO = -i, the i-th argument had an illegal value.
*
*
*  =====================================================================
*
*     .. Parameters ..
      DOUBLE PRECISION   ONE, ZERO
      PARAMETER          ( ONE = 1.0D+0, ZERO = 0.0D+0 )
*     ..
*     .. Local Scalars ..
      INTEGER            n2,n3,n4,j
      DOUBLE PRECISION   temp,tmp2,g,tnorm,inorm,ci
*
*     .. Intrinsics ..
      INTRINSIC          ABS
*
*     .. External Functions ..
      DOUBLE PRECISION   DLANGT
      EXTERNAL           DLANGT

*
*     Test the input arguments.
*
      info = 0
      if( N.LT.0 ) then
         INFO = -2
      end if
      if( INFO.NE.0 ) then
         call XERBLA( 'algorithm2', -INFO )
```

```
         return
      end if
*
*     Computes the 1-norm of A
*
      tnorm = DLANGT( '1', n, b, a, c )
      inorm = 0
*
*     Quick return if possible
*
      if (n.eq.zero) then
         rcond = one
         return
      end if
      IF( tnorm.EQ.ZERO ) THEN
         rcond = zero
         RETURN
      end if
      IF( N.eq.1) THEN
         RCOND = one
         RETURN
      END IF
*
*     Partition Workspace
*
      n2 = 2*n
      n3 = 3*n
      n4 = 4*n
*
*     QR factorization
*
      temp = a(1)
      g = c(1)
      do 10 j = 1, n-1
         call DLARTG(temp,b(j),ci,tmp2,work(j))
         if (work(j) == 0) then
            rcond = 0
            return
         endif
         work(n+j) = -tmp2*(ci*g+tmp2*a(j+1))
         temp = -tmp2*g+ci*a(j+1)
         work(n3+j) = ci
         if (j < n-1) then
            work(n2+j) = tmp2**2*c(j+1)
```

```
            g = ci*c(j+1)
         endif
         if (j > 1) then
            work(n2+j-1) = tmp2*work(n2+j-1)
         endif
         work(n4+j) = tmp2
 10   continue
      work(n) = temp

      if (work(n) == 0) then
         rcond = 0
         return
      endif
      work(n3+n) = 1.0
*
*     Solve linear system
*
      work(n) = work(n3+n)/work(n)
      work(n-1) = (work(n3+n-1)-work(n)*work(n+n-1))/work(n-1)
      do 20 j = n-2, 1, -1
         work(j) = (work(n3+j)-work(j+1)*work(n+j)-work(j+2)*
     $        work(n2+j))/work(j)
 20   continue
*
*     Compute 1-norm of columns of strictly lower triangular part
*
      work(n+n) = 0.0
      work(n+n-1) = ABS(work(n4+n-1)*work(n))
      do 30 j = n-1,2,-1
         work(n+j-1) = (work(n+j)+ABS(work(j)))*ABS(work(n4+j-1))
 30   continue
*
*     Compute 1-norm of columns of upper triangular part
*
      work(n2+1) = 1.0
      do 40 j = 1,n-1
         if (b(j) == 0) then
            work(n2+j+1) = ABS(work(n2+j)*work(j)*c(j)*work(n3+j))
     $           +ABS(work(n3+j))
         else
            work(n2+j+1) = work(n2+j)*ABS(work(n4+j)*c(j)/b(j))+
     $           ABS(work(n3+j))
         endif
 40   continue
```

```
*
*     Add parts together to get the condition number
*
      inorm = work(n+1)+work(n2+1)*ABS(work(1))
      do 50 j = 1, n-1
         temp = ABS(work(n3+j))*work(n+j+1) +
     $         ABS(work(j+1))*work(n2+j+1)
         if (temp > inorm) then
            inorm = temp
         endif
 50   continue

      rcond = (1.0/inorm)/tnorm
*
*     End of DGTCN2
*
      end
```

## B.2 Complex Codes

```
      SUBROUTINE ZGTCN1(N,b,a,c,rcond,work,info)
*
*     .. Scalar Arguments ..
      INTEGER            N, info
      DOUBLE PRECISION   rcond
*     ..
*     .. Array Arguments ..
      COMPLEX*16   a( * ), b( * ), c( * ), WORK( * )
*     ..
*
*     Purpose
*     =======
*
*     ZGTCN1 computes the reciprical of the 1-norm condition number
*     of a complex tridiagonal matrix A using Algorithm 3.2.2
*     adapted for the complex case.
*
*     Arguments
*     =========
*
*     N       (input) INTEGER
*             The order of the matrix A.  N >= 0.
*
*     a       (input) COMPLEX*16 array, dimension (N)
```

```
*             The n diagonal elements of the tridiagonal matrix A.
*
*     b       (input) COMPLEX*16 array, dimension (N-1)
*             The (n-1) subdiagonal elements of the tridiagonal
*             matrix A.
*
*     c       (input) COMPLEX*16 array, dimension (N-1)
*             The (n-1) superdiagonal elements of the tridiagonal
*             matrix A.
*
*     rcond   (output) DOUBLE PRECISION
*             The reciprical of the 1-norm condition number of A.
*
*     WORK    (workspace) COMPLEX*16 array, dimension (6*N)
*
*     INFO    (output) INTEGER
*             INFO = 0 if the matrix is nonsingular.
*             INFO = -i, the i-th argument had an illegal value.
*
*
*  =====================================================================
*
*     .. Parameters ..
      DOUBLE PRECISION   ONE, ZERO
      PARAMETER          ( ONE = 1.0D+0, ZERO = 0.0D+0 )
*     ..
*     .. Local Scalars ..
      COMPLEX*16         temp,g,tmp2
      DOUBLE PRECISION   inorm,tmpc,tnorm,ci
      INTEGER            i,n2,n3,n4,n5
*
*     .. Intrinsics ..
      INTRINSIC          ABS
*
*     .. External Functions ..
      DOUBLE PRECISION   ZLANGT
      EXTERNAL           ZLANGT
*
*     Test the input arguments.
*
      info = 0
      if( N.LT.0 ) then
         INFO = -2
      end if
```

```
      if( INFO.NE.0 ) then
         call XERBLA( 'algorithm1', -INFO )
         return
      end if
*
*     Computes the 1-norm of A
*
      tnorm = ZLANGT('1',n,b,a,c)
      inorm = 0
*
*     Quick return if possible
*
       if (n.eq.zero) then
         rcond = one
         return
      end if
      IF( tnorm.EQ.ZERO ) THEN
         rcond = zero
         RETURN
      end if
      IF( N.eq.1) THEN
         RCOND = one
         RETURN
      END IF
*
*     Partition Workspace
*
      n2 = 2*n
      n3 = 3*n
      n4 = 4*n
      n5 = 5*n
*
*     QR factorization
*
      temp = a(1)
      g = c(1)
      do 10 i = 1, n-1
         call ZLARTG(temp,b(i),ci,tmp2,work(i))
         if (ABS(work(i)) == 0) then
            rcond = 0.0
            return
         endif
         work(n+i) = -conjg(tmp2)*(ci*g+tmp2*a(i+1))
         temp = -conjg(tmp2)*g+ci*a(i+1)
```

```
         work(n3+i) = ci
         if (i < n-1) then
            work(n2+i) = conjg(tmp2)*tmp2*c(i+1)
            g = ci*c(i+1)
         endif
         if (i > 1) then
            work(n2+i-1) = conjg(tmp2)*work(n2+i-1)
         endif
         work(n4+i) = tmp2
 10   continue
      work(n) = temp
      if (ABS(work(n)) == 0) then
         rcond = 0.0
         return
      endif
      work(n3+n) = 1.0

*
*     Solve linear system
*
      work(n) = work(n3+n)/work(n)
      work(n-1) = (work(n3+n-1)-work(n)*work(n+n-1))/work(n-1)
      do 20 i = n-2,1,-1
         work(i) = (work(n3+i)-work(i+1)*work(n+i)-work(i+2)*
     $        work(n2+i))/work(i)
 20   continue
*
*     Compute 1-norm of columns of strictly lower triangular part
*
      work(n5+n) = 0.0
      work(n5+n-1) = ABS(work(n4+n-1)*(work(n)))
      do 30 i = n-1,2,-1
         work(n5+i-1) = (work(n5+i)+ABS(work(i)))*ABS(work(n4+i-1))
 30   continue
      do 35 i = 2,n
         work(n5+i) = work(n5+i)*ABS(work(n3+i-1))
 35   continue
*
*     QL factorization
*
      temp = a(n)
      g = b(n-1)
      do 40 i = n,2,-1
         call ZLARTG(temp,c(i-1),ci,tmp2,work(i))
```

```
         if (ABS(work(i)) == 0) then
            rcond = 0.0
            return
         endif
         work(n+i) = -conjg(tmp2)*(ci*g+tmp2*a(i-1))
         temp = -conjg(tmp2)*g+ci*a(i-1)
         work(n3+i) = ci
         if (i > 2) then
            work(n2+i) = conjg(tmp2)*tmp2*b(i-2)
            g = ci*b(i-2)
         end if
         if (i < n) then
            work(n2+i+1) = conjg(tmp2)*work(n2+i+1)
         end if
         work(n4+i) = tmp2
 40   continue
      work(1) = temp
      if (ABS(work(1)) == 0) then
         rcond = 0.0
         return
      endif
      work(n3+1) = 1.0
*
*     Solve linear system
*
      work(1) = work(n3+1)/work(1)
      work(2) = (work(n3+2)-work(1)*work(n+2))/work(2)
      do 50 i = 3,n
         work(i) = (work(n3+i)-work(i-1)*work(n+i)-work(i-2)*
     $        work(n2+i))/work(i)
 50   continue
*
*     Compute 1-norm of columns of upper triangular part
*
      work(n2+1) = ABS(work(1))
      do 60 i = 1,n-1
         work(n2+i+1) = work(n2+i)*ABS(work(n4+i+1))+ABS(work(i+1))
 60   continue
*
*     Add parts together to get the condition number
*
      inorm = work(n5+n)+work(n2+n)
      do 70 i = 1, n-1
         tmpc = ABS(work(n3+i+1))*ABS(work(n2+i)) +
```

```
     $        work(n5+i)
              if(tmpc > inorm) then
                 inorm = tmpc
              endif
 70   continue
      rcond = (1.0/inorm)/tnorm
*
*     End of ZGTCN1
*
      end




      subroutine ZGTCN2(N,b,a,c,rcond,work,info)
*
*     .. Scalar Arguments ..
      INTEGER            N,info
      DOUBLE PRECISION   rcond
*     ..
*     .. Array Arguments ..
      COMPLEX*16   a( * ), b( * ), c( * ), WORK( * )
*     ..
*
*     Purpose
*     =======
*
*     ZGTCN2 computes the reciprical of the 1-norm condition number
*     of a complex tridiagonal matrix A using Algorithm 3.2.3
*     adapted for the complex case.
*
*     Arguments
*     =========
*
*     N       (input) INTEGER
*             The order of the matrix A.  N >= 0.
*
*     a       (input) COMPLEX*16 array, dimension (N)
*             The n diagonal elements of the tridiagonal matrix A.
*
*     b       (input) COMPLEX*16 array, dimension (N-1)
*             The (n-1) subdiagonal elements of the tridiagonal
*             matrix A.
```

```
*
*     c       (input) COMPLEX*16 array, dimension (N-1)
*             The (n-1) superdiagonal elements of the tridiagonal
*             matrix A.
*
*     rcond   (output) DOUBLE PRECISION
*             The reciprical of the 1-norm condition number of A.
*
*     WORK    (workspace) COMPLEX*16 array, dimension (5*N)
*
*     INFO    (output) INTEGER
*             INFO = 0 if the matrix is nonsingular.
*             INFO = -i, the i-th argument had an illegal value.
*
*
*  =====================================================================
*
*     .. Parameters ..
      DOUBLE PRECISION   ONE, ZERO
      PARAMETER          ( ONE = 1.0D+0, ZERO = 0.0D+0 )
*     ..
*     .. Local Scalars ..
      INTEGER            n2,n3,n4,j
      COMPLEX*16         temp,tmp2,g
      DOUBLE PRECISION   tnorm,inorm,tmpc,ci
*
*     .. Intrinsics ..
      INTRINSIC          ABS
*
*     .. External Functions ..
      DOUBLE PRECISION   ZLANGT
      EXTERNAL           ZLANGT

*
*     Test the input arguments.
*
      info = 0
      if( N.LT.0 ) then
         INFO = -2
      end if
      if( INFO.NE.0 ) then
         call XERBLA( 'algorithm2', -INFO )
         return
      end if
```

```
*
*     Computes the 1-norm of A
*
      tnorm = ZLANGT( '1', n, b, a, c )
      inorm = 0
*
*     Quick return if possible
*
      if (n.eq.zero) then
         rcond = one
         return
      end if
      IF( tnorm.EQ.ZERO ) THEN
         rcond = zero
         RETURN
      end if
      IF( N.eq.1) THEN
         RCOND = one
         RETURN
      END IF
*
*     Partition Workspace
*
      n2 = 2*n
      n3 = 3*n
      n4 = 4*n
*
*     QR factorization
*
      temp = a(1)
      g = c(1)
      do 10 j = 1, n-1
         call ZLARTG(temp,b(j),ci,tmp2,work(j))
         if (ABS(work(j)) == 0) then
            rcond = 0
            return
         endif
         work(n+j) = -conjg(tmp2)*(ci*g+tmp2*a(j+1))
         temp = -conjg(tmp2)*g+ci*a(j+1)
         work(n3+j) = ci
         if (j < n-1) then
            work(n2+j) = conjg(tmp2)*tmp2*c(j+1)
            g = ci*c(j+1)
         endif
```

```
         if (j > 1) then
            work(n2+j-1) = conjg(tmp2)*work(n2+j-1)
         endif
         work(n4+j) = tmp2
 10   continue
      work(n) = temp

      if (ABS(work(n)) == 0) then
         rcond = 0
         return
      endif
      work(n3+n) = 1.0
*
*     Solve linear system
*
      work(n) = work(n3+n)/work(n)
      work(n-1) = (work(n3+n-1)-work(n)*work(n+n-1))/work(n-1)
      do 20 j = n-2, 1, -1
         work(j) = (work(n3+j)-work(j+1)*work(n+j)-work(j+2)*
     $        work(n2+j))/work(j)
 20   continue
*
*     Compute 1-norm of columns of strictly lower triangular part
*
      work(n+n) = 0.0
      work(n+n-1) = ABS(work(n4+n-1)*work(n))
      do 30 j = n-1,2,-1
         work(n+j-1) = (work(n+j)+ABS(work(j)))*ABS(work(n4+j-1))
 30   continue
*
*     Compute 1-norm of columns of upper triangular part
*
      work(n2+1) = 1.0
      do 40 j = 1,n-1
         if (b(j) == 0) then
            work(n2+j+1) = ABS(work(n2+j)*work(j)*c(j)*work(n3+j))
     $           +ABS(work(n3+j))
         else
            work(n2+j+1) = work(n2+j)*ABS(work(n4+j)*c(j)/b(j))+
     $           ABS(work(n3+j))
         endif
 40   continue
*
*     Add parts together to get the condition number
```

```
*
      inorm = work(n+1)+work(n2+1)*ABS(work(1))
      do 50 j = 1, n-1
         tmpc = ABS(work(n3+j))*work(n+j+1) +
     $         ABS(work(j+1))*work(n2+j+1)
         if (tmpc > inorm) then
            inorm = tmpc
         endif
 50   continue

      rcond = (1.0/inorm)/tnorm
*
*     End of ZGTCN2
*
      end
```

# Bibliography


[1] S. T. Alexander, C.-T. Pan, and R. J. Plemmons. Analysis of a recursive least squares hyperbolic rotation algorithm for signal processing. *Linear Algebra and its Applications*, 98:3–40, 1988.

[2] A. A. Anda and H. Park. Fast plane rotations with dynamic scaling. *SIAM J. Matrix Anal. Appl.*, 15:162–174, 1994.

[3] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, third edition, 1999.

[4] Yik-Hoi Au-Yeung, Chi-Kwong Li, and Leiba Rodman. $H$-unitary and Lorentz matrices: A review. *SIAM Journal on Matrix Analysis and Applications*, 25(4):1140–1162, October 2004.

[5] M. Van Barel, D. Fasino, L. Gemignani, and N. Mastronardi. Orthogonal rational functions and structured matrices. *SIAM J. Matrix Anal. Appl.*, 26(3):810–829, 2005.

[6] Marc Van Barel, Ellen Van Camp, and Nicola Mastronardi. Two fast algorithms for solving diagonal-plus-semiseparable linear systems. *J. Comput. Appl. Math.*, 164–165:731–747, 2004.

[7] Dario A. Bini, Luca Gemignani, and Françoise Tisseur. The Ehrlich-Aberth method for the nonsymmetric tridiagonal eigenvalue problem. Numerical Analysis Report No. 428, Manchester Centre for Computational Mathematics, Manchester, England, 2003. To appear in SIAM J. Matrix Anal. Appl.

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

[9] Adam Bojanczyk, Nicholas J. Higham, and Harikrishna Patel. Solving the indefinite least squares problem by hyperbolic QR factorization. *SIAM J. Matrix Anal. Appl.*, 24(4):914–931, 2003.

[10] Adam Bojanczyk, Nicholas J. Higham, and Patel Patel. The constrained indefinite least squares problem: Theory and algorithms. *BIT*, 45(3):505–517, 2003.

[11] Adam Bojanczyk, Sanzheng Qiao, and Allan O. Steinhardt. Unifying unitary and hyperbolic transformations. *Linear Algebra and its Applications*, 316(1–3):183–197, 2000.

[12] A. Bunse-Gerstner. An analysis of the $HR$ algorithm for computing the eigenvalues of a matrix. *Linear Algebra and its Applications*, 35:155–173, 1981.

[13] S. Chandrasekaran and M. Gu. Fast and stable algorithms for banded plus semiseparable matrices. *SIAM J. Matrix. Anal. Appl.*, 25:373–384, 2003.

[14] S. Chandrasekaran, M. Gu, and A. H. Sayed. A stable and efficient algorithm for the indefinite linear least-squares problem. *SIAM J. Matrix Anal. Appl.*, 20(2):354–362, 1999.

[15] S. Chandrasekaran and Ali H. Sayed. Stabilizing the generalized Schur algorithm. *SIAM J. Matrix Anal. Appl.*, 17(4):950–983, 1996.

[16] Philip I. Davies and Nicholas J. Higham. A Schur–Parlett algorithm for computing matrix functions. *SIAM J. Matrix Anal. Appl.*, 25(2):464–485, 2003.

[17] Inderjit Dhillon. Reliable computation of the condition number of a tridiagonal matrix in $O(n)$ time. *SIAM J. Matrix Anal. Appl.*, 19(3):776–796, 1998.

[18] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *LINPACK Users's Guide*. SIAM, 1979.

[19] Y. Eidelman and I. Gohberg. A modification of the Dewilde-van der Veen method for inversion of finite structured matrices. *Linear Algebra and its Applications*, 343–344:419–450, 2001.

[20] F. R. Gantmacher. *The Theory of Matrices*, volume 1. Chelsea, New York, NY, USA, 1959.

[21] P. E. Gill, G. H. Golub, W. Murray, and M. A. Saunders. Methods for modifying matrix factorizations. *Mathematics of Computation*, 28(126):505–535, 1974.

[22] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, Maryland, third edition, 1996.

[23] Joseph F. Grcar. Adjoint formulas for condition numbers applied to linear and indefinite least squares. Submitted to *SIAM J. Matrix Anal. Appl.*

[24] L. Greengard and V. Rokhlin. On the solution of two-point boundary value problems. *Comm. Pure Appl. Math.*, 44:419–452, 1991.

[25] Desmond J. Higham. Condition numbers and their condition numbers. *Linear Algebra and its Applications*, 214:193–213, 1995.

[26] Nicholas J. Higham. *Functions of a Matrix: Theory and Computation.* Book in preparation.

[27] Nicholas J. Higham. The Matrix Computation Toolbox. `http://www.ma.man.ac.uk/~higham/mctoolbox`.

[28] Nicholas J. Higham. Efficient algorithms for computing the condition number of a tridiagonal matrix. *SIAM J. Sci. Statist. Comput.*, 7:150–165, 1986.

[29] Nicholas J. Higham. Evaluating Padé approximants of the matrix logarithm. *SIAM J. Matrix Anal. Appl.*, 22(4):1126–1135, 2001.

[30] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[31] Nicholas J. Higham. $J$-orthogonal matrices: Properties and generation. *SIAM Review*, 45(3):504–519, 2003.

[32] Nicholas J. Higham. The scaling and squaring method for the matrix exponential revisited. *SIAM J. Matrix Anal. Appl.*, 26(4):1179–1193, 2005.

[33] Nicholas J. Higham, D. Steven Mackey, Mackey Mackey, and Françoise Tisseur. Functions preserving matrix groups and iterations for the matrix square root. *SIAM J. Matrix Anal. Appl.*, 26(3):849–877, 2005.

[34] Nicholas J. Higham and Matthew I. Smith. Computing the matrix cosine. *Numerical Algorithms*, 34:13–26, 2003.

[35] IEEE standard for binary floating point arithmetic. Technical Report Std. 754–1985, IEEE/ANSI, New York, 1985.

[36] Sheon-Young Kang, Israel Koltracht, and Rawitscher Rawitscher. Nyström–Clenshaw–Curtis quadrature for integral equations with discontinuous kernels. *Mathematics of Computation*, 72(242):729–756, 2003.

[37] Charles Kenney and Alan J. Laub. Condition estimates for matrix functions. *SIAM J. Matrix Anal. Appl.*, 10(2):191–209, 1989.

[38] June-Yub Lee and Leslie Greengard. A fast adaptive numerical method for stiff two-point boundary value problems. *SIAM Journal on Scientific Computing*, 18(2):403–429, 1997.

[39] A. Magnus and J. Wynn. On the Padé table of $\cos(z)$. *Proc. Amer. Math. Soc.*, 47(2):361–367, 1975.

[40] Nicola Mastronardi, Shivkumar Chandrasekaran, and Sabine Van Huffel. Fast and stable two-way algorithm for diagonal plus semi-separable systems of linear equations. *Numerical Linear Algebra with Applications*, 8(1):7–12, 2001.

[41] R. Onn, A. O. Steinhardt, and A. Bojanczyk. The hyperbolic singular value decomposition and applications. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 39:1575–1588, 1991.

[42] Harikrishna Patel. *Solving the Indefinite Least Squares Problem.* PhD thesis, University of Manchester, 2002.

[43] M. S. Paterson and L. J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM Journal on Computing*, 2(1):60–66, 1973.

[44] Steven M. Serbin. Rational approximations of trigonometric matrices with application to second-order systems of differential equations. *Applied Mathematics and Computation*, 5(1):75–92, 1979.

[45] Steven M. Serbin and Sybil A. Blalock. An algorithm for computing the matrix cosine. *SIAM J. Sci. Statist. Comput.*, 1(2):198–204, 1980.

[46] I. Slapničar. *Accurate symmetric eigenreduction by a Jacobi method.* PhD thesis, Fernuniversität - Hagen, Hagen, Germany, 1992.

[47] Harold P. Starr. *On the numerical solution of one-dimensional integral and differential equations.* PhD thesis, Yale University, 1991.

[48] G. W. Stewart. *Introduction to Matrix Computations.* Academic Press, New York, NY, USA, 1973.

[49] G. W. Stewart. The effects of rounding error on an algorithm for downdating a Cholesky factorization. *Journal of the Institute of Mathematics and its Applications*, 23(2):203–213, 1979.

[50] G. W. Stewart. The efficient generation of random orthogonal matrices with an application to condition estimators. *SIAM Journal on Numerical Analysis*, 17(3):403–409, 1980.

[51] G. W. Stewart. *Matrix Algorithms I: Basic Decompositions.* SIAM, Philadelphia, 1998.

[52] Michael Stewart and G. W. Stewart. On hyperbolic triangularization: Stability and pivoting. *SIAM J. Matrix Anal. Appl.*, 19(4):847–860, 1998.

[53] Françoise Tisseur. Tridiagonal-diagonal reduction of symmetric indefinite pairs. *SIAM J. Matrix Anal. Appl.*, 26(1):215–232, 2004.

[54] C. F. Van Loan. A note on the evaluation of matrix polynomials. *IEEE Transactions on Automatic Control*, AC-24:320–321, 1978.

[55] R. Vandebril, M. Van Barel, and N. Mastronardi. An orthogonal similarity reduction of a matrix to semiseparable form. Technical Report TW355, Katholieke Universiteit Leuven, Belgium, 2003. To appear in SIAM J. Matrix Anal. Appl.

[56] Hongguo Xu. A backward stable hyperbolic QR factorization method for solving indefinite least squares problem. *Journal of Shanghai University (English Edition)*, 8:391–396, 2004.