On Computing Givens Rotations Reliably and Efficiently

DAVID BINDEL, JAMES DEMMEL, and WILLIAM KAHAN
University of California, Berkeley
and
OSNI MARQUES
Lawrence Berkeley National Laboratory

We consider the efficient and accurate computation of Givens rotations. When f and g are positive real numbers, this simply amounts to computing the values of $c = f/\sqrt{f^2 + g^2}$, $s = g/\sqrt{f^2 + g^2}$, and $r = \sqrt{f^2 + g^2}$. This apparently trivial computation merits closer consideration for the following three reasons. First, while the definitions of c, s and r seem obvious in the case of two nonnegative arguments f and g, there is enough freedom of choice when one or more of f and g are negative, zero or complex that LAPACK auxiliary routines SLARTG, CLARTG, SLARGV and CLARGV can compute rather different values of c, s and r for mathematically identical values of f and g. To eliminate this unnecessary ambiguity, the BLAS Technical Forum chose a single consistent definition of Givens rotations that we will justify here. Second, computing accurate values of c, s and r as efficiently as possible and reliably despite over/underflow is surprisingly complicated. For complex Givens rotations, the most efficient formulas require only one real square root and one real divide (as well as several much cheaper additions and multiplications), but a reliable implementation using only working precision has a number of cases. On a Sun Ultra-10, the new implementation is slightly faster than the previous LAPACK implementation in the most common case, and 2.7 to 4.6 times faster than the corresponding vendor, reference or ATLAS routines. It is also more reliable; all previous codes occasionally suffer from large inaccuracies due to over/underflow. For real Givens rotations, there are also improvements in speed and accuracy, though not as striking. Third, the design process that led to this reliable implementation is quite systematic, and could be applied to the design of similarly reliable subroutines.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: *efficiency*; *reliability and robustness*

D. Bindel acknowledges support from a National Science Foundation Graduate Research Fellowship.
J. Demmel acknowledges support from Advanced Research Projects Agency contract no. DAAH04-95-1-0077 (via subcontract no. ORA4466.02 with the University of Tennessee), the Department of Energy grant no. DE-FG03-94ER25219, and contract no. W-31-109-Eng-38 (via subcontract nos. 20552402 and 941322401 with Argonne National Laboratory), the National Science Foundation grants ASC-9313958 and ASC-9813361, and NSF Infrastructure Grant nos. CDA-8722788 and CDA-9401156.
Authors' addresses: D. Bindel, J. Demmel, and W. Kahan: Computer Science Division, University of California, Berkeley, Berkeley, CA, 94720; O. Marques, NERSC, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, MS 50F, Berkeley, CA 94720.

General Terms: Algorithms, Performance, Reliability, Standardization

Additional Key Words and Phrases: BLAS, Givens rotation, linear algebra

1. INTRODUCTION

Givens rotations [Golub and Van Loan 1996; Demmel 1997; Wilkinson 1965] are widely used in numerical linear algebra. Given f and g, a Givens rotation is a 2-by-2 unitary matrix $R(c, s)$ such that

$$R(c,s)\cdot\begin{bmatrix} f \\ g \end{bmatrix} \equiv \begin{bmatrix} c & s \\ -\bar{s} & \bar{c} \end{bmatrix}\cdot\begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix} \tag{1}$$

The fact that $R(c, s)$ is unitary implies

$$\begin{aligned} R(c,s)\cdot(R(c,s))^* &= \begin{bmatrix} c & s \\ -\bar{s} & \bar{c} \end{bmatrix}\cdot\begin{bmatrix} \bar{c} & -s \\ \bar{s} & c \end{bmatrix} \\ &= \begin{bmatrix} c\bar{c}+s\bar{s} & 0 \\ 0 & c\bar{c}+s\bar{s} \end{bmatrix} \\ &= \begin{bmatrix} |c|^2+|s|^2 & 0 \\ 0 & |c|^2+|s|^2 \end{bmatrix} \\ &= I \end{aligned}$$

From this, we see that

$$|c|^2+|s|^2 = 1 \tag{2}$$

When f and g are real and positive, the widely accepted convention is to let

$$\begin{aligned} c &= \frac{f}{\sqrt{f^2+g^2}} \\ s &= \frac{g}{\sqrt{f^2+g^2}} \\ r &= \sqrt{f^2+g^2}. \end{aligned}$$

However, the negatives of c, s and r also satisfy conditions (1) and (2). And when $f = g = 0$, any c and s satisfying (2) also satisfy (1). So c, s and r are not determined uniquely. This slight ambiguity has led to a surprising diversity of inconsistent definitions in the literature and in software. For example, the LAPACK [Anderson et al. 1999] routines SLARTG, CLARTG, SLARGV and CLARGV, the Level 1 BLAS routines SROTG and CROTG [Lawson et al. 1979], as well as Algorithm 5.1.5 in Golub and Van Loan [1996] can get significantly different answers for mathematically identical inputs.

To avoid this unnecessary diversity, the BLAS (Basic Linear Algebra Subroutines) Technical Forum, in its design of the new BLAS standard [Blackford et al. 2001], chose to pick a single definition of Givens rotations. Section 2 below presents and justifies the design.

The BLAS Technical Forum is also providing reference implementations of the new standard. In the case of computing Givens rotations and a few other kernel routines, intermediate over/underflows in straightforward implementations can make the output inaccurate (or stop execution or even cause an infinite loop while attempting to scale the data into a desired range) even though the true mathematical answer might be unexceptional. To compute c, s and r as efficiently as possible and reliably despite over/underflow is surprisingly complicated, particularly for complex f and g.

Square root and division are by far the most expensive real floating point operations on current machines, and it is easy to see that one real square root and one real division (or perhaps a single reciprocal-square-root operation) are necessary to compute c, s and r. With a little algebraic manipulation, we also show that a single square root and division are also sufficient (along with several much cheaper additions and multiplications) to compute c, s and r in the complex case. In contrast, the algorithm in the CROTG routine in the Fortran reference BLAS uses at least five square roots and five divisions, and perhaps 13 divisions, depending on the aggressiveness of the compiler optimizations and implementation of the complex absolute value function cabs.

However, the formulas for c, s and r that use just one square root and one division are susceptible to over/underflow, if we must store all intermediate results in the same precision as f and g. Define $\|f\| = \max(|\texttt{re}\, f|, |\texttt{im}\, f|)$; $|f|$ will denote the usual absolute value of a complex number. We systematically identify the values of f and g for which these formulas are reliable (i.e., guaranteed not to underflow in such a way that unnecessarily loses relative precision, nor to overflow) by generating a set of simultaneous linear inequalities in $\log\|f\|$ and $\log\|g\|$, which define a 2D polygonal region called *Safe* in $(\log\|f\|, \log\|g\|)$ space in which the formulas may be used. This is the most common situation, which we call Case 1 in the algorithm. In this case, the new algorithm runs slightly faster than LAPACK's CLARTG routine, and 2.7 to 4.6 times faster than the CROTG routine in the vendor BLAS, ATLAS BLAS, or Fortran reference BLAS on a Sun Ultra-10.

If $(\log\|f\|, \log\|g\|)$ lies outside *Safe*, there are two possibilities: scaling f and g by a constant to fit inside *Safe*, or using different formulas. Scaling may be interpreted geometrically as shifting *Safe* parallel to the diagonal line $\log\|f\| = \log\|g\|$ in $(\log\|f\|, \log\|g\|)$ space. The region covered by shifted images of *Safe* (*Safe*'s "shadow") is the region in which scaling is possible. In part of this shadow (case 4 in the algorithm), we do scale f and g to lie inside *Safe* and then use the previous formula.

The remaining region of $(\log\|f\|, \log\|g\|)$ space, including space outside *Safe*'s shadow, consists of regions where $\log\|f\|$ and $\log\|g\|$ differ so much that $|f|^2 + |g|^2$ rounds either to $|f|^2$ (Case 2 in the algorithm) or to $|g|^2$ (Case 3). Replacing $|f|^2+|g|^2$ by either $|f|^2$ or $|g|^2$ simplifies the algorithm, and different formulas are used.

In addition to the above four cases, there are two other simpler ones, when f and/or g is zero.

There are three different ways to deal with these multiple cases. The first way is to have tests and branches depending on $\|f\|$ and $\|g\|$ so that only the appropriate formula is used. This is the most portable method, using only working precision (the precision of the input/output arguments) and is the one explored in most detail in this paper.

The second method is to use exception handling, that is, assume that f and g fall in the most common case (Case 1), use the corresponding formula, and only if a floating point exception is raised (overflow, underflow, or invalid) is an alternative formula used [Demmel and Li 1994]. If sufficiently fast exception handling is available, this method may be fastest.

The third method assumes that a floating point format with a wider exponent range is available to store intermediate results. In this case we may use our main new formula (Case 1) without fear of over/underflow, greatly simplifying the algorithm (the cases of f and/or g being zero remain). For example, IEEE double precision (with an 11-bit exponent) can be used when inputs f and g are IEEE single precision numbers (with 8-bit exponents). On a Sun Ultra-10, this mixed-precision algorithm is nearly exactly as fast in Case 1 of the single precision algorithm described above, and usually rather faster in Cases 2 through 4. On an Intel machine, double-extended floating point (with 15-bit exponents) can be used for single- or double-precision inputs, and this would be the algorithm of choice. However, with double-precision inputs on a machine like a Sun Ultra-10 without double-extended arithmetic, or when double precision is much slower than single precision, our new algorithm with four cases is the best we know.

A similar approach to implementing complex elementary functions was taken in Hull et al. [1994; 1997].

In addition to the new algorithm being significantly faster than previous routines, it is more accurate. All earlier routines have inputs that exhibit large relative errors, whereas ours is always accurate.

The rest of this article is organized as follows: Section 2 presents and justifies the proposed definition of Givens rotations. Section 3 details the differences between the proposed definition and existing LAPACK and Level 1 BLAS code. Section 4 describes our assumptions about floating point arithmetic. Section 5 presents the algorithm in the complex case, for the simple cases when $f = 0$ or $g = 0$. Section 6 presents the algorithm in the most common complex case, assuming that neither overflow nor underflow occur (Case 1). Section 7 shows alternate formulas for complex Givens rotations when f and g differ greatly in magnitude (Cases 2 and 3). Section 8 describes scaling when f and g are comparable in magnitude but both very large or very small (Case 4). Section 9 summarizes the overall algorithm for complex Givens rotations. Section 10 compares the accuracy of our new complex Givens routine and several alternatives; only ours is accurate in all cases. Section 11 discusses the performance of our complex Givens routine. Sections 12, 13, and 14 discuss algorithms, accuracy and timing for real Givens rotations, which are rather easier. Section 15 draws conclusions. The actual software is available in Bindel et al. [2002] and Blackford et al. [2002].

2. DEFINING GIVENS ROTATIONS

We use the following function, defined for a complex variable x, in what follows:

$$\operatorname{sign}(x) \equiv \begin{cases} x/|x| & \text{if } x \neq 0 \\ 1 & \text{if } x = 0 \end{cases}$$

sign(x) is clearly a continuous function away from $x = 0$. When x is real the definition simplies to

$$\operatorname{sign}(x) \equiv \begin{cases} -1 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0. \end{cases}$$

As stated in the introduction, we need extra requirements besides (1) and (2) in order to determine c and s (and hence r) uniquely. For given any c, s and r satisfying (1) and (2), so do $\omega \cdot c$, $\omega \cdot s$, and $\omega \cdot r$, where ω is any complex number with unit absolute value. (When $f = g = 0$, one clearly has more freedom of choice since (1) is automatically satisfied.) In this section we discuss how to choose ω. In particular, when f and g are not both zero, one can see that the set of possible choices for c, s and r is

$$\begin{aligned} c &= \omega \cdot \frac{\bar{f}}{\sqrt{|f|^2 + |g|^2}} \\ s &= \omega \cdot \frac{\bar{g}}{\sqrt{|f|^2 + |g|^2}} \\ r &= \omega \cdot \sqrt{|f|^2 + |g|^2}. \end{aligned} \tag{3}$$

2.1 Reasonable Desiderata for Choosing ω

We list below a number of reasonable desiderata for a choice of ω, show that they cannot all be attained simultaneously, and that they in fact trade off against one another. We discuss several design alternatives, notably one in Anderson [2000], to argue for our design. Here are the desiderata:

—**Compatibility** with existing applications. Few if any routines using current implementations of Givens rotations should break.

—**Consistency** between definitions for real and complex rotations: they should agree on real data. In particular c, s and r should be real if f and g are real.

—**Continuity**. The values of c, s and r should be continuous functions of the inputs f and g, if possible.

—**Efficiency**. The definition should permit a fast implementation.

Compatibility is paramount: we cannot propose a definition that breaks existing algorithms, and expect anyone to adopt it. In particular, this means that c must be real even when f and g are complex, since this is current practice in the existing BLAS [Lawson et al. 1979], EISPACK [Smith et al. 1976], LINPACK [Dongarra et al. 1979] and LAPACK [Anderson et al. 1999]. (This prior choice of c real was motivated by efficiency: it takes less time to apply a complex Givens rotation if we know c is real.) When f is nonzero making c real means ω

must have one of two values: $\mathrm{sign}(f)$ (so that $c > 0$) or $-\mathrm{sign}(f)$ (so that $c < 0$). In other words, we have to choose the sign bit of c.

When $f = 0$ and $g \neq 0$, so that $c = 0$, the choice of ω is not limited by the compatibility constraint.

Consistency is attractive because it makes the definition shorter and simpler, and makes it likely that one will get the same answer when applying a complex algorithm to real data as a real algorithm. In other words, it helps minimize surprises.

When $f = 0$ and $g \neq 0$ is real, consistency implies that $\omega = \pm 1$.

Continuity is harder to formulate precisely. Note that c and s must be discontinuous at least at $(f, g) = (0, 0)$, since c and s have fixed different absolute values along different straight lines passing through $(0, 0)$. So we must somehow be able to measure whether one function is "more continuous" than another.

Furthermore, there are actually six functions whose continuity we consider (c, s and r as functions of f and g, where f and g are either real or complex). It is not clear if it is more important for one of these functions to be continuous than another.

We have chosen the following approach to measure continuity. Consider the real function $s(f, g)$ of two real variables. Consider the set $D \subset \mathbf{R}^2$ of points $(f, g) \in \mathbf{R}^2$ where $s(f, g)$ is discontinuous. We can measure the size of D by its **dimension** as a subset of $\mathbf{R}^2$: a finite set of point has dimension $\dim(D) = 0$, a finite set of curves has dimension $\dim(D) = 1$, and so on (the sets D we encounter will be quite simple, so we will not need the more sophisticated notion of Hausdorff dimension). If we have two functions s_1 and s_2 with sets of discontinuities D_1 and D_2, then we will say s_1 is *more continuous* than s_2 if $\dim(D_1) < \dim(D_2)$. The intuition is that a finite set of points is smaller than a finite set of curves, which is in turn smaller than a 2D surface, etc.

Now suppose f and g are complex, so that $s(f, g)$ is a function of 4 real variables (the real and imaginary parts of f and g). We define the set of discontinuities D as above, as well as its *real* (as opposed to *complex*) dimension. For example, the set $D = \{(f, g) : f = 0\}$ has (real) dimension 2, and the set $D = \{(f, g) : \mathtt{re}\, f = 0\}$ has (real) dimension 3. Henceforth, when we say dimension we mean real dimension.[1]

Finally, recall that we have six functions whose continuity we must consider. To measure the discontinuity of the real Givens rotation, we take all three functions c, s and r of two real variables, compute the dimensions of their sets of discontinuity, and take the maximum of these three dimensions. We may similarly measure the discontinuity of the complex Givens rotation. Finally, we

[1]We note that we could instead compare the *codimensions* of the sets D_1 and D_2 instead of their dimensions, saying that D_1 is smaller than D_2 if it has *larger codimension* instead of smaller dimension. (The codimension of D is the dimension of the ambient space containing D minus $\dim(D)$.) If D_1 and D_2 are subsets of the same ambient space it does not matter whether we compare dimensions or codimensions, but if D_1 and D_2 lie in different ambient spaces, then it may be possible to say that either D_1 or D_2 is smaller. For example, consider $D_1 = \{(0, 0)\} \subset \mathbf{R}^2$ and $D_2 = \{(f, 0) : \mathtt{re}\, f = 0\} \subset \mathbf{C}^2 = \mathbf{R}^4$. D_1 has both smaller dimension and smaller codimension than D_2. For simplicity, we choose to compare dimensions in this paper.

take the maximum of these two dimensions. This is the measure of discontinuity we would like to minimize.

We note that a different approach is taken in [Anderson 2000], and we discuss this below.

2.2 Consequences of Our Desiderata

Having defined our desiderata carefully, we may proceed to see how well they may be satisfied; we omit proofs which are simple computations.

Away from the origin $(f, g) = (0, 0)$, formulas (3) are continuous functions of f and g if and only if $\omega = \omega(f, g)$ is a continuous function; $\omega \equiv 1$ is just the simplest choice. But compatibility (keeping c real) means that, if $f \neq 0$, then we must have $\omega(f, g) = \sigma(f, g) \cdot \mathrm{sign}(f)$, where $\sigma(f, g) \in \{+1, -1\}$. This function is necessarily discontinuous in the complex case at $f = 0$, since $\mathrm{sign}(f)$ takes on all possible values in its range in an arbitrarily small neighborhood of $f = 0$. The set $\{(f, g) \;:\; f = 0\}$ has dimension 2. Therefore, compatibility means we cannot maximize continuity.

2.2.1 *Maximizing Continuity of Complex Givens Rotations.*

Let us just try to maximize continuity in the complex case, that is, minimize the maximum dimension of the sets of discontinuity of c, s and r. Dimension 2 is the best we can hope for. Keeping c, s and r real and positive when f and g are real and positive means $\sigma(f, g) \equiv 1$ or $\omega(f, g) = \mathrm{sign}(f)$ or

$$\begin{aligned} c &= \frac{|f|}{\sqrt{|f|^2 + |g|^2}} \\ s &= \mathrm{sign}(f) \cdot \frac{\bar{g}}{\sqrt{|f|^2 + |g|^2}} \\ r &= \mathrm{sign}(f) \cdot \sqrt{|f|^2 + |g|^2} \end{aligned} \tag{4}$$

when $f \neq 0$. It remains to define c, s and r when $f = 0$. When $f = 0$ and $g \neq 0$, we (somewhat arbitrarily) choose ω to make c, s and r continuous functions of f and $g \neq 0$ as f approaches zero through positive real numbers; this means $\omega(0, g) = 1$ and so

$$\begin{aligned} c &= 0 \\ s &= \mathrm{sign}(\bar{g}) \\ r &= |g|, \end{aligned} \tag{5}$$

when $f = 0$ and $g \neq 0$. Finally, when $f = g = 0$, we cannot hope to choose c and s by continuity arguments, so we appeal to efficiency to justify the choice $c = 1$ and $s = 0$.

In summary, the algorithm that maximizes continuity of c, s and r for complex f and g is as follows (the dimension of the set of discontinuities is 2):

Algorithm 1: Computing Givens Rotations

```
if $g = 0$ (includes the case $f = g = 0$)
    $c = 1$
    $s = 0$
    $r = f$
```

```
elseif $f = 0$ ($g$ must be nonzero)
    $c = 0$
    $s = \text{sign}(\bar{g})$
    $r = |g|$
else ($f$ and $g$ both nonzero)
    $c = |f|/\sqrt{|f|^2 + |g|^2}$
    $s = \text{sign}(f)\bar{g}/\sqrt{|f|^2 + |g|^2}$
    $r = \text{sign}(f)\sqrt{|f|^2 + |g|^2}$
endif
```

2.2.2 *Maximizing Continuity of Real Givens Rotations.*. It is easy to see that the choice $\omega(f, g) \equiv 1$ makes r continuous everywhere, and c and s continuous everywhere except their unavoidable zero-dimensional discontinuity at $(f, g) = (0, 0)$. However, Algorithm 1 uses the choice $\omega(f, g) = \text{sign}(f)$, which introduces a discontinuity at $f = 0$, a set of dimension 1 (since this is the real case).

In other words, consistency (using the same definition for real and complex data) means that we cannot maximize continuity in the real case. Alternatively, abandoning consistency means that we can maximize continuity for real and for complex rotations simultaneously.

2.2.3 *Maximizing Total Continuity.* Using Algorithm 1 for both real and complex Givens rotations satisfies compatibility (c is real), consistency (same definition for real and complex data), and maximizes overall continuity, because the maximum dimension of the set of discontinuities is at most 2, and this cannot be improved. (The dimension is 1 in the real case, which could be reduced to 0, but 2 in the complex case, which cannot be reduced, so the maximum is an unimprovable 2.)

2.3 Anderson's Alternative

We discuss a different choice of $\omega(f, g)$ proposed by Anderson [2000]: $\omega(f, g) = \text{sign}(f) \cdot \text{sign}(\texttt{re}\, f)$. Note that in the real case this simplifies to $\omega = 1$, which maximizes continuity in the real case. However, it creates a discontinuity of dimension 3 at $\{(f, g) : \texttt{re}\, f = 0\}$ in c, s and r in the complex case.

Anderson's main motivation in keeping the real Givens rotation as continuous as possible is to address the desires of users of LAPACK's symmetric eigenvalue subroutines for "continuous behavior," that is, for computed eigenvectors to change only slightly when the input matrix changes only slightly.[2]

This is not always possible, since when eigenvalues are (nearly) multiple very small changes in the matrix can cause large changes in their associated eigenvectors. But when eigenvalues are "well separated" (far from multiple), so that the spaces spanned by eigenvectors are in fact continuous functions of the matrix entries [Kato 1980], it seems like a more reasonable request. Anderson observed that by replacing LAPACK's (also discontinuous) `slartg` routine for

[2]Anderson also states that continuity is necessary for algorithms like the QR algorithm for eigenvalues that use Givens rotations to be backward stable, but this is not the case; all the algorithms we know of in LAPACK or elsewhere assume only that (1) and (2) are satisfied (to within round-off) to guarantee stability.

computing real Givens rotations by the continuous definition described above, eigenvectors frequently behaved continuously when they did not before.

A complete discussion of eigenvalue and eigenvector computations is beyond the scope of this article, but we argue here that the users' desire for continuous output is (1) theoretically impossible, and (2) can be better approximated by a different approach than choosing a particular definition of Givens rotations.

First, while an invariant subspace corresponding to an isolated eigenvalue may be a uniquely defined continuous function of the matrix entries, its associated eigenvector is not, simply because eigenvectors are not unique: if v is an eigenvalue, so is βv for any scalar $\beta \neq 0$. Without knowing the sequence of nearby matrices at which the eigenvector routine is to be called, there is no unique way to define v in a continuous way. In the literature on continuous matrix decompositions [Kunkel and Mehrmann 1991], one assumes one has a matrix function $A(t)$, and defines eigenvectors that are continuous along this path, that is, one knows the set of nearby matrices. In other words, as long as one matrix at a time is handed to the eigenvalue algorithm, it is not possible to define the eigenvectors continuously.

Second, it is not enough to make the Givens (and Householder) transformations as continuous as possible in the QR or other algorithms for eigenvalues [Golub and Van Loan 1996; Demmel 1997], because it is easy to concoct examples where intermediate computations occur at discontinuities of these transformations. For example, if entries 2 through n of the first row and column of the matrix are zero, arbitrarily small perturbations in these entries will lead to an *arbitrary* first Householder transformation in the reduction to tridiagonal form. It is generally not nearly so obvious from looking at the input matrix whether such an intermediate discontinuity can occur, so that occasional discontinuities can still surprise the user. A better alternative is to define a normalization (choose β and output βv) whose (dis)continuity is easy to understand. One natural way (*not* currently done in LAPACK) is to choose β so that βv has unit Euclidean norm *and* the component of largest magnitude is real and positive. If there is a tie, choose the component of lowest index. This clearly can still be discontinuous; consider an eigenvector with two different components of largest magnitude and opposite signs: a small matrix perturbation can make either one the unique largest component and so swap the sign of β. Any analogous rule must also have a discontinuity.

The above arguments for choosing our definition over Anderson's alternative are not overwhelming. For example, if continuity in real rotations were much more important than for complex rotations in a particular application, we would prefer Anderson's definition.

2.4 Exceptional Cases

When this algorithm is run on a computer with IEEE floating point arithmetic [ANSI/IEEE 1985] it is possible that some inputs might be NaNs (Not-a-Number symbols) or $\pm\infty$. In this section we discuss the values c, s and r should have in these cases. We describe both what an implementor of the new BLAS standard [Blackford et al. 2001] *should* do on a machine that correctly implements IEEE floating point semantics with NaNs and $\pm\infty$ (some decisions are left to the discretion of the implementor) as well as what our implementation does do.

On a machine with correct IEEE floating point semantics with NaNs and $\pm\infty$, we insist that any standard conforming implementation must terminate and return some output values in all cases as described below. If the machine does not implement these semantics correctly, for example by trapping and stopping execution as some machines do in "fast mode," then we can make no claims about what the algorithm should or will do. The rest of this section assumes correct IEEE floating point semantics.

We say that a complex number is a NaN if at least one of its real and imaginary parts is a NaN. We say that a complex number is infinite if at least one of its real and imaginary parts is infinite, and neither is a NaN.

First, suppose at least one NaN occurs as input. The IEEE semantics of NaN are that any binary or unary arithmetic operation on a NaN returns a NaN, so that by extension our routine ought to return NaNs as well. But we see that our definition above will not necessarily do this, since if $g = 0$ an implementation might reasonably still return $c = 1$ and $s = 0$ even if f is a NaN, since these require no arithmetic operations to compute. Rather than specify exactly what should happen when an input is NaN, we insist only that in a standard conforming implementation *at least* r must be a NaN; c and s may also be NaNs, at the implementor's discretion. We permit this discretion because NaNs are (hopefully!) very rare in most computations, and insisting on testing for this case might slow down the code too much in common cases. Our implementation will only guarantee that r is a NaN.

To illustrate the challenges of correct portable coding with NaNs, consider computing $\max(a, b)$, which we will need to compute $\|f\|$ and $\|g\|$. If max is implemented (in hardware or software) as "if $(a > b)$, then a, else b" then $\max(0, \mathrm{NaN})$ returns NaN but $\max(\mathrm{NaN}, 0)$ returns 0. On the other hand, the equally reasonable implementation "if $(a < b)$, then b, else a" instead returns 0 and NaN, respectively.[3] Thus, an implementation might mistakenly decide $g = 0$ because $\|g\| = 0$ and then return $c = 1$, $s = 0$ and $r = f$, missing the NaN in g. Indeed, if g has exactly one component equal to NaN, $\|g\|$ may consistently return the absolute value of the other component; thus in *every* path of the code based on testing $\|f\|$ and $\|g\|$, we must consider the possibility that f and g are actually NaNs. (We encountered this during development; the value returned by max may depends the level of compiler optimization.) Our model implementation will work with any implementation of max.

Next, suppose at least one ∞ or $-\infty$ occurs as input, but no NaNs. In this case, it is reasonable to return the limiting values of the definition if they exist,

[3]The IEEE floating point standard does not specify the semantics of max and min, so these behaviors are both consistent with the standard. This ambiguity about the meaning of max and min is being taken up by both the IEEE standards committee considering a revision of the IEEE floating point standard, and the Java Grande Working Group. It turns out that statisticians use NaN to mean "missing data" and strongly prefer $\max(3, \mathrm{NaN}) = 3$, though we prefer NaN. The Java Grande Working Group and IEEE 754 Floating Point Standard Revision Group [IEEE 754-2002] are considering proposals to support both kinds of max function.

or NaNs otherwise. For example, one might return $c = 0$, $s = 1$ and $r = \infty$ if $f = 0$ and $g = \infty + i \cdot 0$ but $c = 0$, $s = \text{NaN}$ and $r = \infty$ if $f = 0$ and $g = \infty + i \cdot \infty$ since $s = \text{sign}(\bar{g}) = \bar{g}/g$ cannot be well defined while $r = |g|$ can be. Or one could simply return NaNs even if a limit existed, for example, returning $c = 0$, $s = \text{NaN}$ and $r = \text{NaN}$ whenever $f = 0$ and $\|g\| = \infty$. Again to avoid overspecifying rare cases and thereby possibly slowing down the common cases, we leave it to the standard implementor's discretion which approach to take. But we insist that at least r either be infinite or a NaN. This is all our implementation guarantees.

The assiduous reader will have noted that we have not explicitly discussed how the sign of zero is to be treated (IEEE arithmetic includes both $+0$ and -0). Algorithm 1 uses $\text{sign}(0) = 1$, but a standard conforming implementor would be free to use the copysign routine, which would return $\text{sign}(-0) = -1$ in the real case. There seems to be little to be gained by insisting, for example, that $r = -0$ when $f = -0$ and $g = -0$, which is what would actually be computed if $R(1, +0)$ were multiplied by the vector $[-0, -0]^T$.

3. DIFFERENCES FROM CURRENT LAPACK AND BLAS CODES

Here is a short summary of the differences between Algorithm 1 and the algorithms in LAPACK 3.0 [Anderson et al. 1999] and earlier versions, and in the Level 1 BLAS [Lawson et al. 1979]. The LAPACK algorithms in question are SLARTG, CLARTG, SLARGV and CLARGV, and the Level 1 BLAS routines are SROTG and CROTG.

The current LAPACK subroutines SLARTG and CLARTG (which compute a single real and complex Givens rotation, respectively) are not consistent (they can give quite different results on the same real-valued inputs). Furthermore, the LAPACK subroutines SLARGV and CLARGV for computing multiple Givens rotations do not compute the same answers as SLARTG and CLARTG, respectively, (This discrepancy was also pointed out in Anderson and Fahey [1997].) The differences are described below. So some change in practice is needed to have consistent definitions. (Indeed, this was the original motivation for the BLAS Technical Forum not simply adopting the LAPACK definitions unchanged.)

All the LAPACK release 3.0 test code passed as well with the new Givens rotations as with the old ones (indeed, one test failure in the old code disappeared with the new rotations), so the new definition of Givens rotations satisfies our compatibility requirement.

SLARTG. When $f = 0$ and $g \neq 0$, Algorithm 1 returns $s = \text{sign}(g)$ whereas SLARTG returns $s = 1$. The comment in SLARTG about "saving work" does not mean the LAPACK bidiagonal SVD routine SBDSQR assumes $s = 1$. When $|f| \leq |g|$ and $f < 0$ (so both f and g are nonzero), SLARTG returns the negatives of the values of c, s and r returned by Algorithm 1.

CLARTG. Algorithm 1 is mathematically identical to CLARTG. But it is not numerically identical, see Section 10 below.

SLARGV. When $f = g = 0$, SLARGV returns $c = 0$ and $s = 1$ instead of $c = 1$ and $s = 0$. When $f \neq 0$ and $g = 0$, SLARGV returns $c = \text{sign}(f)$ instead of $c = 1$. When $f = 0$ and $g \neq 0$, SLARGV returns $s = 1$ instead of $s = \text{sign}(g)$. When $f \neq 0$ and $g \neq 0$, SLARGV returns $\text{sign}(c) = \text{sign}(f)$, instead of $c \geq 0$.

CLARGV. When $f = g = 0$, CLARGV returns $c = 0$ and $s = 1$ instead of $c = 1$ and $s = 0$. When $f = 0$ and $g \neq 0$, CLARGV returns $s = 1$ instead of $s = \text{sign}(\bar{g})$.

SROTG. SROTG overwrites f by r and g by a quantity z from which one can reconstruct both s and c [Stewart 1976] ($z = s$ if $|f| > |g|$, $z = 1/c$ if $|g| \geq |f|$ and $c \neq 0$, and $z = 1$ otherwise). Besides this difference, r is assigned the sign of g as long as either f or g is nonzero, rather than the sign of f (or 1).

CROTG. CROTG overwrites f by r, but does not compute a quantity like z. CROTG sets $c = 0$ and $s = 1$ if $f = 0$, rather than $s = \text{sign}(\bar{g})$ if $g \neq 0$ and $c = 1, s = 0$ if both $f = g = 0$. When both f and g are nonzero, it matches Algorithm 1 mathematically, but not numerically.

4. ASSUMPTIONS ABOUT FLOATING POINT ARITHMETIC

In LAPACK, we have the routines SLAMCH and DLAMCH available, which return various machine constants that we will need. In particular, we assume that ε = machine epsilon is available, which is a power of the machine radix. On machines with IEEE floating point arithmetic [ANSI/IEEE 1985], it is either 2^{-24} in single or 2^{-53} in double. Also, we use `SAFMIN`, which is intended to be the smallest normalized power of the radix whose reciprocal can be computed without overflow. On IEEE machines, this should be the underflow threshold, 2^{-126} in single and 2^{-1022} in double. However, on machines where complex division is implemented in the compiler by the fastest but risky algorithm

$$\frac{a+ib}{c+id} = \frac{ac+bd}{c^2+d^2} + i\frac{bc-ad}{c^2+d^2}$$

the exponent range is effectively halved, since $c^2 + d^2$ can over/underflow even though the true quotient is near 1. On these machines `SLAMCH` may set `SAFMIN` to $\sqrt{\texttt{SAFMIN}}$ to indicate this. As a result, our scaling algorithms make no assumptions about the proximity of `SAFMIN` to the actual underflow threshold, and indeed any tiny value rather less than ε will lead to correct code, though the closer `SAFMIN` is to the underflow threshold the fewer scaling steps are needed in extreme cases.

Our algorithms also work correctly and accurately whether or not underflow is gradual. This is important on the processors where default "fast mode" replaces all underflowed quantities by zero. This means that we assume that underflow can introduce an error as large as `SAFMIN`, so that only if a computed quantity y is at least $\texttt{SAFMIN}/\varepsilon$ in magnitude can we be sure that underflow in x can cause a relative error in $x + y$ of at most ε.

This means that $\texttt{SAFMIN}/\varepsilon$ is an important quantity in our scaling algorithm. In fact we will use the quantity $(\varepsilon/\texttt{SAFMIN})^{1/4}$ rounded up to the nearest power of the radix; we call this rounded quantity z. Thus, we use $z^{-4} \approx \texttt{SAFMIN}/\varepsilon$ as the effective underflow threshold, and $z^4 = \varepsilon/\texttt{SAFMIN}$ as the overflow threshold. Note that we may safely add and subtract many quantities bounded in magnitude by z^4 without incurring overflow. We repeat that the algorithms work correctly, if more slowly, if a conservative estimate of SAFMIN is used (i.e., one that is too large). The powers of z used by the software are computed on the first call, and then saved and reused for later calls. On a fixed platform these powers of z could be constants hardwired into the code. The values of z and its powers for IEEE machines with SAFMIN equal to the underflow threshold are as follows:

	Single Precision	Double Precision
SAFMIN	$2^{-126} \approx 1 \cdot 10^{-38}$	$2^{-1022} \approx 2 \cdot 10^{-308}$
ε	$2^{-24} \approx 6 \cdot 10^{-8}$	$2^{-53} \approx 1 \cdot 10^{-16}$
z	$2^{25} \approx 3 \cdot 10^{7}$	$2^{242} \approx 7 \cdot 10^{72}$
z^4	$2^{100} \approx 1 \cdot 10^{30}$	$2^{968} \approx 2 \cdot 10^{291}$
z^{-1}	$2^{-25} \approx 3 \cdot 10^{-8}$	$2^{-242} \approx 1 \cdot 10^{-73}$
z^{-4}	$2^{-100} \approx 7 \cdot 10^{-31}$	$2^{-968} \approx 4 \cdot 10^{-292}$

When inputs include $\pm\infty$ and NaN, we assume the semantics of IEEE arithmetic [ANSI/IEEE 1985] are used. In Section 10, we discuss the impact of compiler optimization flags on this assumption.

In later discussions we denote the actual overflow threshold by OV, the underflow threshold by UN, and the smallest positive number by m, which is $2 \cdot \varepsilon \cdot$UN on a machine with gradual underflow, and UN otherwise.

5. COMPLEX ALGORITHM WHEN $f=0$ OR $g=0$

In what follows, we use the convention of capitalizing all variable names, so that C, S and R are the data to be computed from F and G. We use the notation re(F) and im(F) to mean the real and imaginary parts of F, and $\|w\| = \max(|\mathrm{re}\, w|, |\mathrm{im}\, w|)$ for any complex number w. We begin by eliminating the easy cases where at least one of F and G is zero. Variables F, G, S and R are complex, and the rest are real.

Algorithm 2: Computing Givens Rotations when $f=0$ or $g=0$

```
if G = 0
    ... includes the case F = G = 0
    C = 1
    S = 0
    R = F
else if F = 0
    ... G must be nonzero
    C = 0
    scale G by powers of $z^{\pm 4}$ so that $z^{-2} \le \|G\| \le z^2$
    D1 = sqrt(re(G)**2+im(G)**2)
    R = D1
    D1 = 1/D1
    S = conj(G)*D1
    unscale R by powers of $z^{\pm 4}$
else
    ... both F and G are nonzero
    ... use algorithm described below
endif
```

We note that even though $F = 0 \neq G$ is an "easy" case we need to scale G to avoid over/underflow when computing `re(G)**2+im(G)**2`.

5.1 Exceptional Cases

Now we discuss exception handling. It noticeably speeds up the code to implement the tests `G = 0` and `F = 0` by precomputing `SG =` $\|G\|$ and `SF =` $\|F\|$, which will be used later, and then testing whether `SG = 0` and `SF = 0`. But as described in Section 2.4, either of these tests might succeed even though the real or imaginary part of F or G is a NaN. Therefore, the logic of the algorithm must change slightly as shown below.

Algorithm 2E: Computing Givens Rotations when $f = 0$ or $g = 0$, with exception handling

```
SG = $\|G\|$
SF = $\|F\|$
if SG = 0
    ... includes the case F = G = 0
    C = 1
    S = 0
    ... In case G is a NaN, make sure R is too
    R = F+G
else if SF = 0
    ... G must be nonzero
    C = 0
    scale G by powers of $z^{\pm 4}$ so that $z^{-2} \le \|G\| \le z^2$
        ... limit number of scaling steps in case G infinite or NaN
    D1 = sqrt(re(G)**2+im(G)**2)
    R = D1
    D1 = 1/D1
    S = conj(G)*D1
    unscale R by powers of $z^{\pm 4}$
    ... In case F is a NaN, make sure R is too
    R = R + F
else
    ... both F and G are nonzero
    ... use algorithm described below
endif
```

The test `SG=0` can succeed if one part of G is 0 and the other is a NaN, which is why we must return `R = F+G` instead of `R = F` to make sure the input NaN propagates to the output R. Note that outputs `C=1` and `S=0` even if there are NaNs and infinities on input.

Similarly, the branch where `SF = 0` can be taken when G is a NaN or infinity. This means that a loop to scale G (and SG) into range might not terminate if written without an upper bound on the maximum number of steps it can take. This maximum is essentially $\max(\lceil \log_z \mathrm{OV} \rceil, -\lfloor \log_z m \rfloor)$. The timing depends strongly on implementation details of scaling (use of unrolling, loop structure, etc.). The algorithm we used could probably be improved by tuning to a particular compiler and architecture. C will always be zero, but S will be a NaN if G is either infinite or a NaN, and R will be infinite precisely if G is infinite.

6. COMPLEX ALGORITHM WHEN *F* AND *G* ARE NONZERO

Now assume F and G are both nonzero. We can compute C, S and R with the following code fragment, which employs only one division and one square root. The last column shows the algebraically exact quantity computed by each line of code. We assume that real$*$complex multiplications are performed by two real multiplications (the Fortran implementation does this explicitly rather than relying on the compiler). Variables `F`, `G`, `R` and `S` are complex, and the rest are real.

Algorithm 3: Fast Complex Givens Rotations when *f* and *g* are "well scaled"

```
1.    F2   : = re(F)**2 + im(F)**2    $|f|^2$
2.    G2   : = re(G)**2 + im(G)**2    $|g|^2$
3.   FG2   : = F2 + G2                $|f|^2+|g|^2$
4.    D1   : = 1/sqrt(F2*FG2)         $1/\sqrt{|f|^4+|f|^2|g|^2}$
                                        $= 1/(|f|\sqrt{|f|^2+|g|^2})$
5.     C   : = F2*D1                  $|f|/\sqrt{|f|^2+|g|^2}$
6.   FG2   : = FG2*D1                 $\sqrt{|f|^2+|g|^2}/|f|$
                                        $= \sqrt{1+|g|^2/|f|^2}$
7.     R   : = F*FG2                  $f\sqrt{1+|g|^2/|f|^2}$
                                        $= \mathrm{sign}(f)\sqrt{|f|^2+|g|^2}$
8.     S   : = F*D1                   $\frac{f}{|f|}\frac{1}{\sqrt{|f|^2+|g|^2}}$
9.     S   : = conj(G)*S              $\frac{f}{|f|}\frac{\bar{g}}{\sqrt{|f|^2+|g|^2}}$
```

Now recall $z=(\varepsilon/\texttt{SAFMIN})^{1/4}$, so that z^4 is an effective overflow threshold and z^{-4} is an effective underflow threshold. The region where the above algorithm can be run reliably is described by the following inequalities, which are numbered to correspond to lines in the above algorithm. All logarithms are to the base 2.

(1) We assume $\|f\| \le z^2$ to prevent overflow in computation of `F2`.

(2) We assume $\|g\| \le z^2$ to prevent overflow in computation of `G2`.

(3) This line is safe given previous assumptions.

(4a) We assume $z^{-2} \le \|f\|$ to prevent underflow of `F2` and consequent division by zero in the computation of `D1`.

(4b) We assume $\|f\| \le z$ to prevent overflow from the $|f|^4$ term in `F2*FG2` in the computation of `D1`.

(4c) We assume $\|f\|\|g\| \le z^2$ to prevent overflow from the $|f|^2|g|^2$ term in `F2*FG2` in the computation of `D1`.
Either (4d) $z^{-1} \le \|f\|$
or (4e) $z^{-2} \le \|f\|\|g\|$
to prevent underflow of `F2*FG2` and consequent division by zero in the computation of `D1`.

(5) This line is safe given previous assumptions. If `C` underflows, it is deserved.

(6) $\|g\|/\|f\| \le z^4$ to prevent overflow of `FG2` since $\sqrt{1+|g|^2/|f|^2} = O(|g|/|f|)$ if $|g|/|f|$ is large.

(7) This line is safe given previous assumptions, returning $|\texttt{R}|$ roughly between z^{-1} and z^2. If the smaller component of `R` underflows, it is deserved.

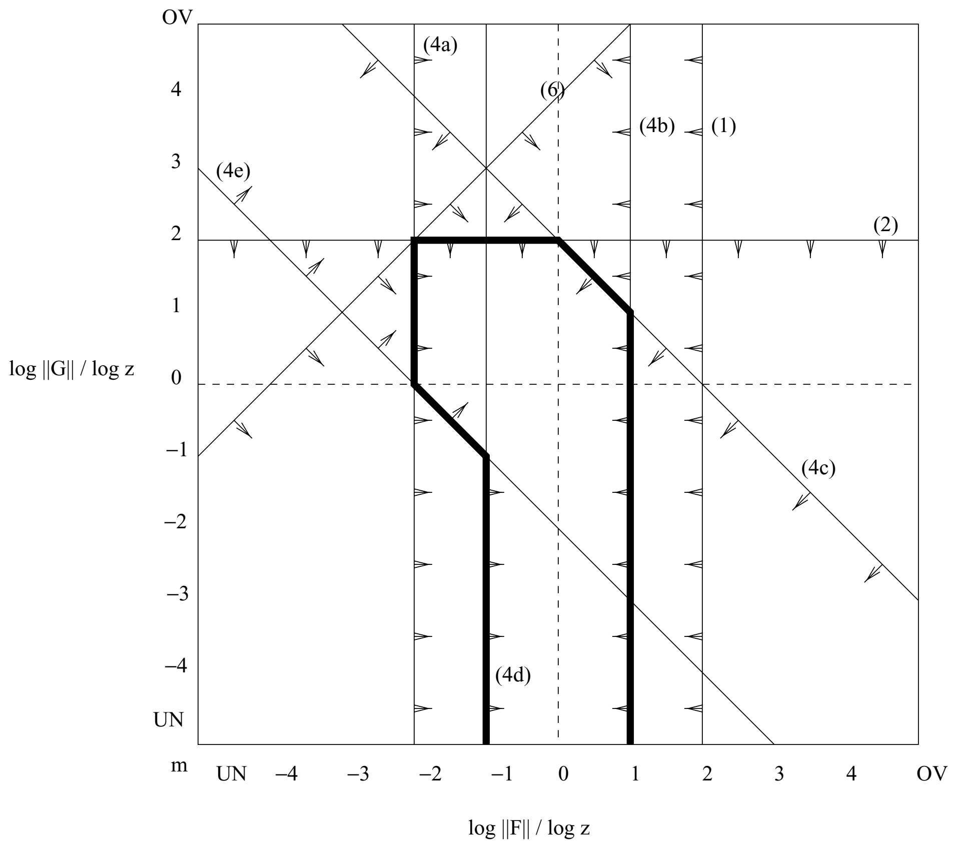

Fig. 1. Inequalities describing the region of no unnecessary over/underflow. UN and OV are the over/underflow thresholds; m is the smallest representable positive number.

(8) This line is safe given previous assumptions, returning |S| roughly between z^{-2} and 1. The smaller component of S may underflow, but this error is very small compared to the other component of S.

(9) This line is safe given previous assumptions. If S underflows, it is deserved.

All the inequalities in the above list describe half planes in $(\log \| f \|, \log \| g \|)$ space. For example, inequality (6) becomes $\log \| g \| - \log \| f \| \leq 4 \log z$.

The region described by all inequalities is shown in Figure 1. Each inequality is described by a thin line marked by arrows indicating the side on which the inequality holds. The heavy line borders the safe region S satisfying all the inequalities, where the above algorithm can be safely used.

It remains to say how to decide whether a point lies in *Safe*. The boundary of *Safe* is complicated, so the time to test for membership in *Safe* can be nontrivial. Accordingly, we use the simplest tests that are likely to succeed first, and only then do we use more expensive tests. In particular, the easiest tests are threshold comparisons with $\| f \|$ and $\| g \|$, and the most common case is when $\| f \|$ and $\| g \|$ are neither very large nor very small. So we test for membership in the subset of *Safe* labeled (1) in Figure 2 by the following algorithm:

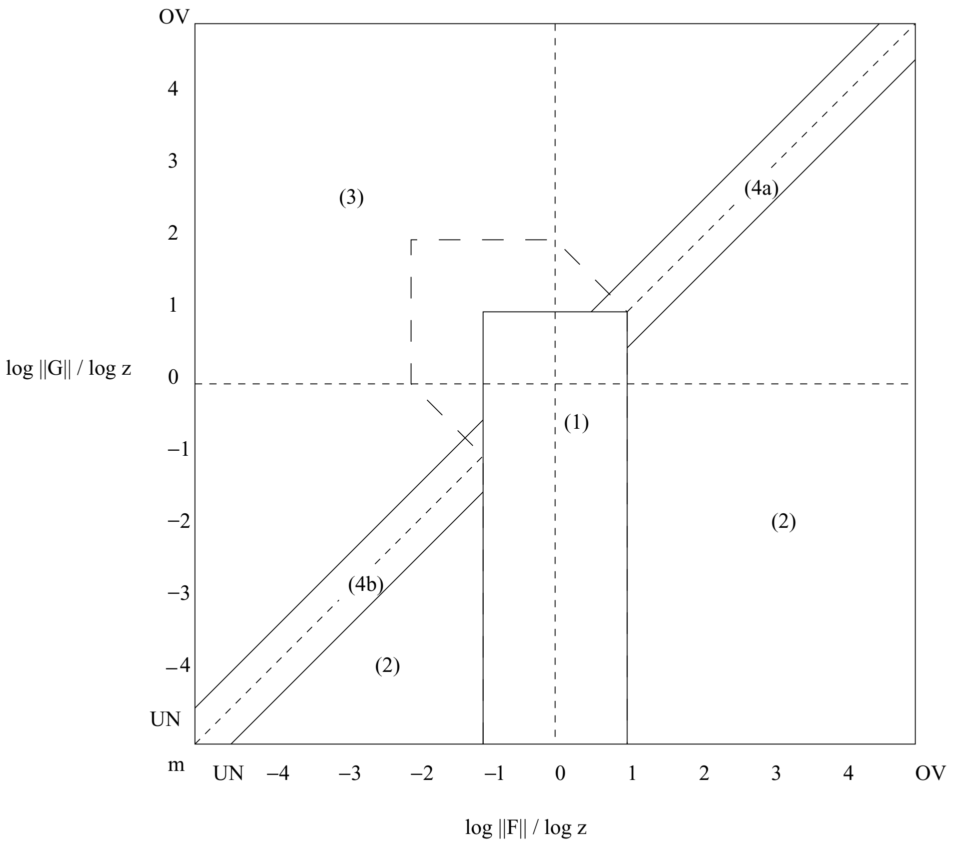

Fig. 2. Cases in the code when $f \neq 0$ and $g \neq 0$.

```
if $\|f\| \leq z$ and $\|f\| \geq z^{-1}$ and $\|g\| \leq z$ then
    $f, g$ is in Region (1)
endif
```

This is called Case 1 in the software.

Region (1) contains all data where $\|f\|$ and $\|g\|$ are not terribly far from 1 in magnitude (between $z^{\pm 1} = 2^{\pm 25} \approx 10^{\pm 7}$ in single and between $z^{\pm 1} = 2^{\pm 242} \approx 10^{\pm 73}$ in double), which we expect to include most arguments, especially in double.

The complement of Region (1) in *Safe* is shown bounded by dashed lines in Figure 2. It is harder to test for, because its boundaries require doing threshold tests on the product $\|f\| \cdot \|g\|$, which could overflow. So we will not test for membership in this region explicitly in this case, but do something else instead. (We could easily test for membership in the square with opposite corners (0,0) and (−2,2), but it is simpler to cover these cases as described in Section 7 below.)

6.1 Exceptional Cases

Again we consider the consequence of NaNs and infinities. It is easy to see that if either `F` or `G` is infinite, then the above test for membership in Region (1) cannot succeed. So it suffices to consider NaNs.

Any test like `A` $\geq$ `B` evaluates to false when either `A` or `B` is a NaN, so Case 1 occurs with NaN inputs only when $\|f\|$ and $\|g\|$ are not NaNs, which can occur as described in Section 2.4. By examining Algorithm 3 we see that a NaN in `F` or `G` leads to `FG2` and then all of `C`, `S` and `R` being NaNs.

7. COMPLEX ALGORITHM WHEN f AND g DIFFER GREATLY IN MAGNITUDE

When $|g|^2 \leq \varepsilon |f|^2$, then $|f|^2 + |g|^2$ rounds to $|f|^2$, and the formulas for c, s and r may be greatly simplified and very accurately approximated by

$$\begin{aligned} c &\approx 1 \\ s &\approx \operatorname{sign}(f)\frac{\bar{g}}{|f|} = \frac{f \cdot \bar{g}}{|f|^2} \\ r &\approx f \end{aligned} \tag{6}$$

This region is closely approximated by the two regions $\|g\| \leq \varepsilon^{1/2}\|f\|$ bounded by solid lines and marked (2) in Figure 2, and is called Case 2 in the software.

When instead $|f|^2 \leq \varepsilon |g|^2$, then $|f|^2 + |g|^2$ rounds to $|g|^2$, and the formulas for c, s and r may be greatly simplified and very accurately approximated by

$$\begin{aligned} c &\approx \frac{|f|}{|g|} = \frac{|f|^2}{|f| \cdot |g|} \\ s &\approx \operatorname{sign}(f)\frac{\bar{g}}{|g|} = \frac{f \cdot \bar{g}}{|f| \cdot |g|} \\ r &\approx \operatorname{sign}(f)|g| = \frac{f \cdot |g|^2}{|f| \cdot |g|} \end{aligned} \tag{7}$$

This region is closely approximated by the region $\|f\| \leq \varepsilon^{1/2}\|g\|$ bounded by solid lines and marked (3) in Figure 2, and is called Case 3 in the software.

An important difference between the formulas in (6) and (7) versus the formula (4) is that (6) and (7) are independently homogeneous in f and g. In other words, we can scale f and g *independently* instead of by the same scalar in order to evaluate them safely. Thus, the "shadow" of the region in which the above formulas are safe covers all (f, g) pairs. In contrast in formula (4) f and g must be scaled by the same value.

Here are the algorithms implementing (6) and (7) without scaling. Note that (6) does not even require a square root.

Algorithm 4: Computing complex Givens rotations when $\|g\| \leq \sqrt{\varepsilon}\|f\|$, using formulas (6), without scaling

```
if $\|G\| < \sqrt{\varepsilon} \cdot \|F\|$ then
    C = 1
    R = F
    D1 = 1/(re(F)**2 + im(F)**2)
    S = F * conj(G)
    S = S * D1
endif
```

Algorithm 5: Computing complex Givens rotations when $\|g\| \le \sqrt{\varepsilon}\|f\|$, using formulas (7), without scaling

```
if $\|F\| < \sqrt{\varepsilon} \cdot \|G\|$ then
    F2  = re(F)**2 + im(F)**2
    G2  = re(G)**2 + im(G)**2
    FG2 = F2 * G2
    D1  = 1/sqrt(FG2)
    C   = F2 * D1
    S   = F * conj(G)
    S   = S * D1
    D1  = D1 * G2
    R   = D1 * F
endif
```

We may now apply the same analysis as in the last section to these formulas, deducing linear inequalities in $\log\|f\|$ and $\log\|g\|$ which must be satisfied in order to guarantee safe and accurate execution. We simply summarize the results here. In both cases, we get regions with boundaries that, like *Safe*, are sets of line segments that may be vertical, horizontal or diagonal. We again wish to restrict ourselves to tests on $\|f\|$ and $\|g\|$ alone, rather than their product (which might overflow). This means that we identify a smaller safe region (like region (1) within *Safe* in Figure 2) where membership can be easily tested. This safe region for Algorithm 4 is the set satisfying

$$z^{-2} \le \|f\| \le z^2 \quad \text{and} \quad z^{-2} \le \|g\| \le z^2 \tag{8}$$

This safe region for Algorithm 5 is the smaller set satisfying

$$z^{-1} \le \|f\| \le z \quad \text{and} \quad z^{-1} \le \|g\| \le z \tag{9}$$

This leads to the following algorithms, which incorporate scaling.

Algorithm 6: Computing complex Givens rotations when $\|g\| \le \sqrt{\varepsilon}\|f\|$, using formulas (6), with scaling

```
if $\|G\| < \sqrt{\varepsilon} \cdot \|F\|$ then
    C = 1
    R = F
    scale F by powers of $z^{\pm 4}$ so $z^{-2} \le \|F\| \le z^2$
    scale G by powers of $z^{\pm 4}$ so $z^{-2} \le \|G\| \le z^2$
    D1 = 1/(re(F)**2 + im(F)**2)
    S = F * conj(G)
    S = S * D1
    unscale S by powers of $z^{\pm 4}$ to undo scaling of F and G
end if
```

Algorithm 7: Computing complex Givens rotations when $\|f\| \le \sqrt{\varepsilon}\|g\|$, using formulas (7), with scaling

```
if $\|F\| < \sqrt{\varepsilon} \cdot \|G\|$ then
    scale F by powers of $z^{\pm 2}$ so $z^{-1} \le \|F\| \le z$
    scale G by powers of $z^{\pm 2}$ so $z^{-1} \le \|G\| \le z$
    F2  = re(F)**2 + im(F)**2
    G2  = re(G)**2 + im(G)**2
    FG2 = F2 * G2
    D1  = 1/sqrt(FG2)
    C   = F2 * D1
    S   = F * conj(G)
    S   = S * D1
    D1  = D1 * G2
    R   = D1 * F
    unscale C and R by powers of $z^{\pm 2}$ to undo scaling of F and G
endif
```

Note in Algorithm 7 that the value of S is unaffected by independent scaling of F and G.

7.1 Exceptional cases

First, consider Case 2, that is, Algorithm 6. It is possible for either F or G to be NaNs (since $\|F\|$ and $\|G\|$ may not be) but if neither is a NaN then only F can be infinite (since the test is $\|G\| < \sqrt{\varepsilon} \cdot \|F\|$, not $\|G\| \leq \sqrt{\varepsilon} \cdot \|F\|$). Care must be taken as before to assure termination of the scaling of F and G even when they are NaNs or infinite.

In Case 2, C = 1 independent of whether inputs are infinite or NaNs. S is a NaN if either F or G is a NaN or infinite. If we simply set R=F then R is a NaN (or infinite) precisely when F is a NaN (or infinite); in other words R might not be a NaN if G is. So in our model implementation we can and do ensure that R is a NaN if either F or G is a NaN by instead computing R = F + S*G after computing S.

Next consider Case 3, that is, Algorithm 7. Analogous comments about the possible values of the inputs as above apply, and again care must be taken to assure termination of the scaling. In Case 3, if either input is a NaN, all three outputs will be NaNs. If G is infinite and F is finite, then S and R will be NaNs.

8. COMPLEX ALGORITHM: SCALING IN REGIONS (4a) AND (4b)

For any point (f, g) that does not lie in regions (1), (2) or (3) of Figure 2, we can use the following algorithm:

(1) Scale (f, g) to a point $(scale \cdot f, scale \cdot g)$ that does lie in S.
(2) Apply Algorithm 3 to $(scale \cdot f, scale \cdot g)$, yielding $c, s, \hat{r}$.
(3) Unscale to get $r = \hat{r}/scale$.

This scaling in Figure 2 corresponds to shifting f, g parallel to the diagonal line $f = g$ by $\log scale$ until it lies in *Safe*. It is geometrically apparent that the set of points scalable in regions (4a) and (4b) of Figure 2 lies in the set of all diagonal translates of *Safe*, that is, the "shadow" of *Safe*, and can be scaled to lie in *Safe*. Indeed, all points in region (2) and many (but not all) points in region (3) can be scaled to lie in *Safe*, but in regions (2) and (3) cheaper formulas discussed in the last section are available.

First, suppose that (f, g) lies in region (4a). Let $s = \max(\|f\|, \|g\|)$. Then if $s > z^2$, we can scale f and g down by z^{-2}. Eventually (f, g) will lie in the union of the two arrow-shaped regions A1 and A2 in Figure 3. Then, if s still exceeds z, that is, (f, g) is in A1, we multiply f and g by z^{-1}, putting (f, g) into A2.

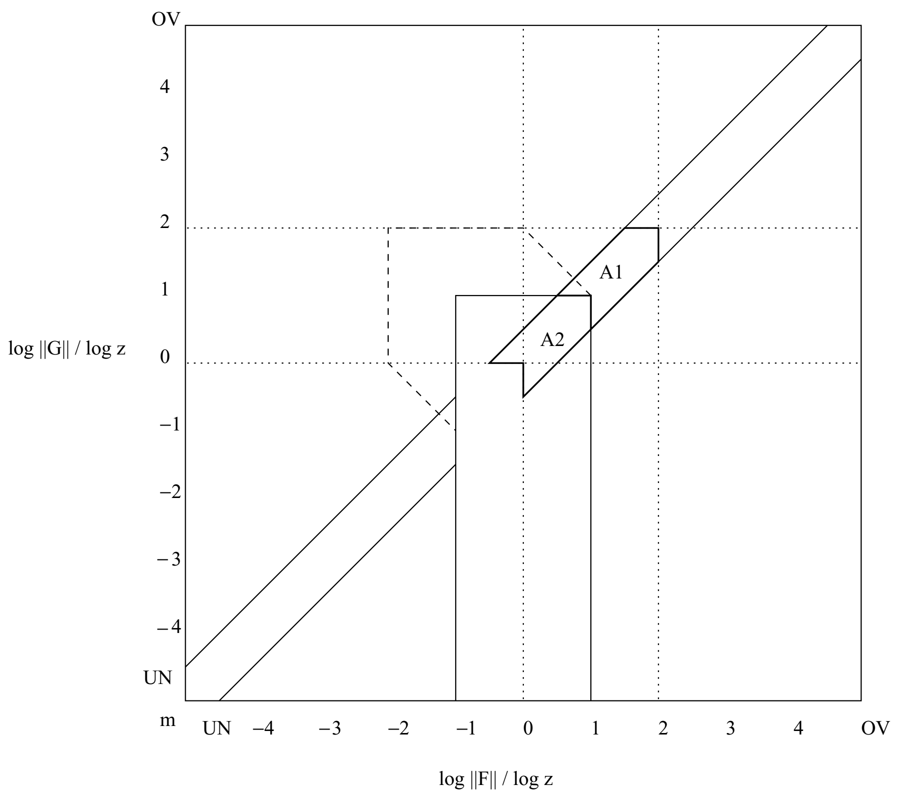

Fig. 3. Scaling when (f, g) is in Region (4a).

Thus, we guarantee that the scaled f and g are in A2, where it is safe to use Algorithm 3.

Next suppose that (f, g) lies in region (4b). Now let $s = \| f \|$. Then, if $s < z^{-2}$, we can scale f and g up by z^2. Eventually, (f, g) will lie in the union of the two parallelograms B1 and B2 in Figure 4. Then, if s is still less than z^{-1}, that is, (f, g) is in B1, we multiply f and g by z, putting (f, g) into B2. Thus, we guarantee that the scaled f and g are in B2, where it is safe to use Algorithm 3.

These considerations lead to the following algorithm

Algorithm 8: Computing complex Givens rotations when (f, g) is in region (4a) or (4b), with scaling.

```
... this code is only executed if $f$ and $g$ are in region (4a) or (4b)
if $\|F\| > 1$
    scale F and G down by powers of $z^{-2}$ until $\max(\|F\|, \|G\|) \le z^2$
    if $\max(\|F\|, \|G\|) > z$, scale F and G down by $z^{-1}$
else
    scale F and G up by powers of $z^2$ until $\|F\| \ge z^{-2}$
    if $\|F\| < z^{-1}$, scale F and G up by $z$
endif
compute the Givens rotation using Algorithm 3
undo the scaling of R caused by scaling of F and G
```

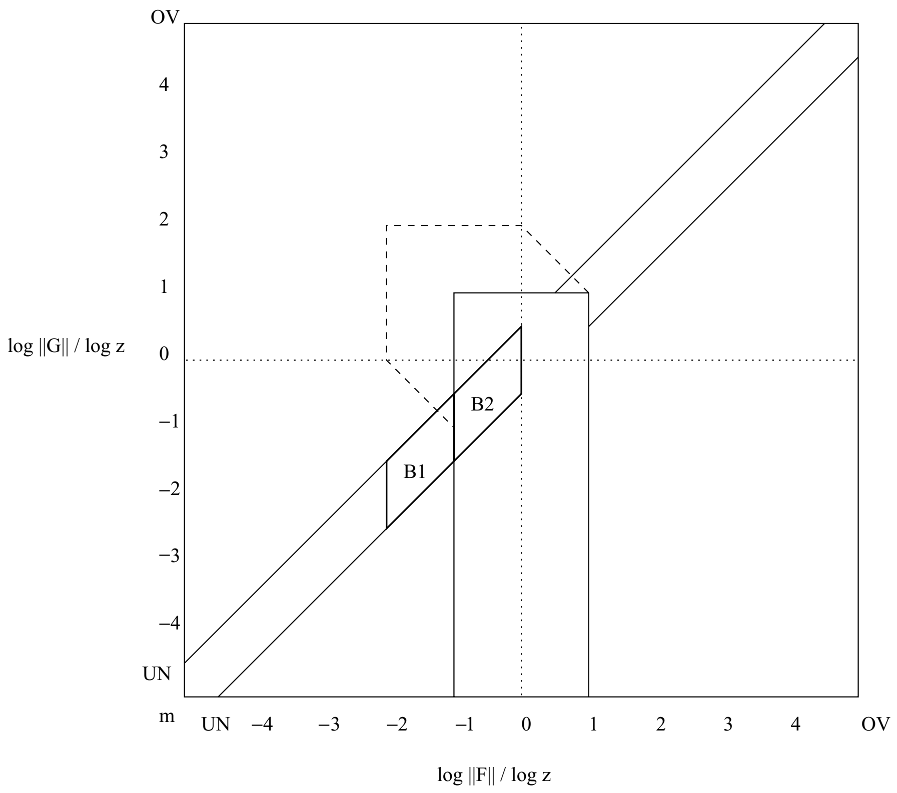

Fig. 4. Scaling when (f, g) is in Region (4b).

8.1 Exceptional cases

Either input may be a NaN, as discussed before. And because this case is selected by "falling through" the logic selecting the previous cases, it is possible that the arguments are simultaneously infinite. In any of these cases, all three outputs will be NaNs. As before, care must be taken in scaling to avoid infinite loops.

9. OVERALL COMPLEX ALGORITHM

We call the overall algorithm new CLARTG, to distinguish from old CLARTG, which is part of the LAPACK 3.0 release. The entire source code is available in Bindel et al. [2002] and Blackford et al. [2001]. It contains 268 noncomment lines, as opposed to 20 in the reference CROTG implementation. Here is how all the algorithms discussed so far fit together.

Algorithm New CLARTG: Computing a complex Givens Rotation from inputs *f* and *g*.

```
Precompute and store $z$ and its powers as described in Section 4
Compute $\|f\|$ and $\|g\|$
If $\|f\| = 0$ or $\|g\| = 0$
  Use Algorithm 2E
Elseif $\|f\| \le z$ and $\|f\| \ge z^{-1}$ and $\|g\| \le z$ then
```

```
  ... Case 1
  Use Algorithm 3
Elseif $\|g\| \le \sqrt{\varepsilon} \cdot \|f\|$ then
  ... Case 2
  Use Algorithm 6
Elseif $\|f\| \le \sqrt{\varepsilon} \cdot \|g\|$ then
  ... Case 3
  Use Algorithm 7
Else
  ... Case 4
  Use Algorithm 8
Endif
```

We note that the boundaries of the regions are "conservative," in the sense that the formulas in each region work well slightly outside each region. So roundoff in the tests for region membership is harmless.

10. ACCURACY RESULTS FOR COMPLEX GIVENS ROTATIONS

The algorithm was run for $49^4 = 5764801$ values of f and g, where the real and imaginary parts of f and g independently took on 49 different values ranging from 0 to the overflow threshold, with intermediate values chosen just above and just below the threshold values determining all the edges and corners in Figures 1 through 4, and thus barely satisfying (or not satisfying) all possible branches in the algorithm. (Another 1.5 million inputs containing infinities and NaNs were also tested for correct behavior.) The correct answer was computed using a straightforward implementation of Algorithm 1 using double precision arithmetic, in which no overflow nor underflow is possible for the arguments tested. The maximum errors in r, c and s were computed as follows. Here r_s was computed in single using the new algorithm and r_d was computed straightforwardly in double precision; the subscripted c and s variables have analogous meanings. In the absence of gradual underflow, the error metric for finitely representable r_s is

$$\frac{|r_s - r_d|}{\max(\varepsilon |r_d|, \texttt{SAFMIN})} \tag{10}$$

and with gradual underflow it is

$$\frac{|r_s - r_d|}{\max(\varepsilon |r_d|, \texttt{SAFMIN} * 2 * \varepsilon)} \tag{11}$$

with the maximum taken over all nonzero test cases for which the true r does not overflow. (On this subset, the mathematical definitions of c, s and r used in CLARTG and CROTG agree). Note that $\texttt{SAFMIN} * 2 * \varepsilon$ is the smallest denormalized number. Analogous metrics were computed for s_s and c_s.

The routines were first tested on a Sun Ultra-10 using f77 (version SUNWspro-6.0) with the -fast -O5 flags, the most aggressive level of compiler optimization, which means gradual underflow is *not* used (i.e., results less than SAFMIN are replaced by 0) and some transformations that do not preserve IEEE semantics may be used.[4] Therefore, we expect the measure (10) to be at least 1,

[4]For example, when dividing a complex number a by a real number b, the reciprocal of b is computed

and hopefully just a little bigger than 1, meaning that the error $|r_s - r_d|$ is either just more than machine epsilon ε times the true result, or a small multiple of the underflow threshold, which is the inherent uncertainty in the arithmetic.

The routines were also tested without any optimization flags, which means exact IEEE floating point compliance is guaranteed, including gradual underflow, so we expect the more stringent measure (11) to be close to 1.

The results are as follows. The explanation of the various lines in the table follows.

Without Gradual Underflow			
Routine	Max error in r_s	Max error in s_s	Max error in c_s
New CLARTG	3.20	4.02	2.45
Old CLARTG	$5 \cdot 10^6$	$5 \cdot 10^6$	$4 \cdot 10^6$
Reference CROTG	NaN	NaN	NaN
Modified Reference CROTG	5.39	4.57	4.12
ATLAS CROTG	NaN	NaN	NaN
Limited ATLAS CROTG	3.13	$2 \cdot 10^7$	2.90
Vendor CROTG	NaN	NaN	NaN
Limited Vendor CROTG	3.57	$2 \cdot 10^7$	3.15

With Gradual Underflow			
Routine	Max error in r_s	Max error in s_s	Max error in c_s
New CLARTG	3.20	3.28	2.57
Old CLARTG	3.45	3.94	$4 \cdot 10^6$
Reference CROTG	NaN	NaN	NaN
Modified Reference CROTG	$7 \cdot 10^6$	$7 \cdot 10^6$	$7 \cdot 10^6$

Here is why the old CLARTG fails to be accurate. First, consider the situation without gradual underflow. When $|g|$ is just above z^{-2}, and $|f|$ is just below, the algorithm will decide that scaling is unnecessary. As a result $|f|^2$ may have a nonnegligible relative error from underflow, which creates a nonnegligible relative error in r, s and c. Now consider the situation with gradual underflow. The above error does not occur, but a different one occurs. When $1 \gg |g| \gg |f|$, and f is denormalized, then the algorithm will not scale. As a result $|f|$ suffers a large loss of relative accuracy when it is rounded to the nearest denormalized number, and then $c \approx |f|/|g|$ has the same large loss of accuracy.

Here is why the reference BLAS CROTG can fail, even though it tries to scale to avoid over/underflow. The scale factor $|f| + |g|$ computed internally can overflow even when $|r| = \sqrt{|f|^2 + |g|^2}$ does not. Now consider the situation without gradual underflow. If $|r|$ does not overflow but is large enough so that its reciprocal underflows to zero, and the compiler optimizes $f/|r|$ by replacing it by $f \cdot (1/|r|)$, we can get zero instead of a number as large as 1. Even if this does not go wrong, the sine can be inaccurate for the following reason. The sine

and multiplied by a, which can lead to spurious underflow. We wanted to make sure our algorithm is impervious to such optimizations.

is computed as $s = (f/|f|)\cdot(\bar{g})/(\sqrt{|f|^2+|g|^2})$, where the multiplication is done first. All three quantities in parentheses may be quite accurate, but the entries of $f/|f|$ are both less than one, causing the multiplication to underflow to 0, when the true s exceeds 0.4. This can be repaired by inserting parentheses $s = (f/|f|)\cdot((\bar{g})/(\sqrt{|f|^2+|g|^2}))$ so the division is done first. Excluding these cases where $|f|+|g|$ overflows or $1/(|f|+|g|)$ underflows, and inserting parentheses, we get the errors on the line "Modified Reference CROTG." Now consider the situation with gradual underflow. Then rounding intermediate quantities to the nearest denormalized number can cause large relative errors, such as s and c both equaling 1 instead of $1/\sqrt{2}$.

The ATLAS (version 3.0beta) and vendor (SUNWspro-6.0) versions of CROTG were only run with the full optimizations suggested by their authors, which means gradual underflow was not enabled. They also return NaNs for large arguments even when the true answer should have been representable. We did not modify these routines, but instead ran them on the limited subset of examples where $|f|+|g|$ was less than overflow (see the lines "Limited ATLAS CROTG" and "Limited Vendor CROTG"). They still occasionally had large errors from underflow causing s to have large relative errors, even when the true value of s is quite large.

In summary, our systematic procedure produced a provably reliable implementation whereas there are errors in all previous implementations that yield inaccurate results without warning, or fail unnecessarily due to overflow. The latter only occurs when the true r is close to overflow, and so it is hard to complain very much, but the former problem deserves to be corrected.

11. TIMING RESULTS FOR COMPLEX GIVENS ROTATIONS

Timings were done on a Sun Ultra-10 using the f77 compiler (version SUNWspro-6.0) with optimization flags -fast -O5. Each routine was called 10^3 times for arguments throughout the f, g plane (see Figure 2) and the average time taken for each argument (f, g); the range of timings for (f, g) was typically only a few percent. Twenty-nine cases were tried in all, exercising all paths in the new CLARTG code. The input data are shown in a table below.

We note that the timing results for optimized code are not entirely predictable from the source code. For example, small changes in the way scaling is implemented can make large differences in the timings. If proper behavior in the presence of infinity or NaN inputs were not an issue (finite termination and propagating infinities and NaNs to the output) then scaling and some other parts of the code could be simplified and probably accelerated.

The timing results are in the Figures 5 and 6. Seven algorithms are compared:

(1) New CLARTG is the algorithm presented in this report, using tests and branches to select the correct case.
(2) OLD CLARTG is the algorithm in LAPACK 3.0.
(3) Ref CROTG is the reference BLAS.
(4) ATLAS CROTG is the ATLAS BLAS.
(5) Vendor CROTG is Sun's vendor BLAS.

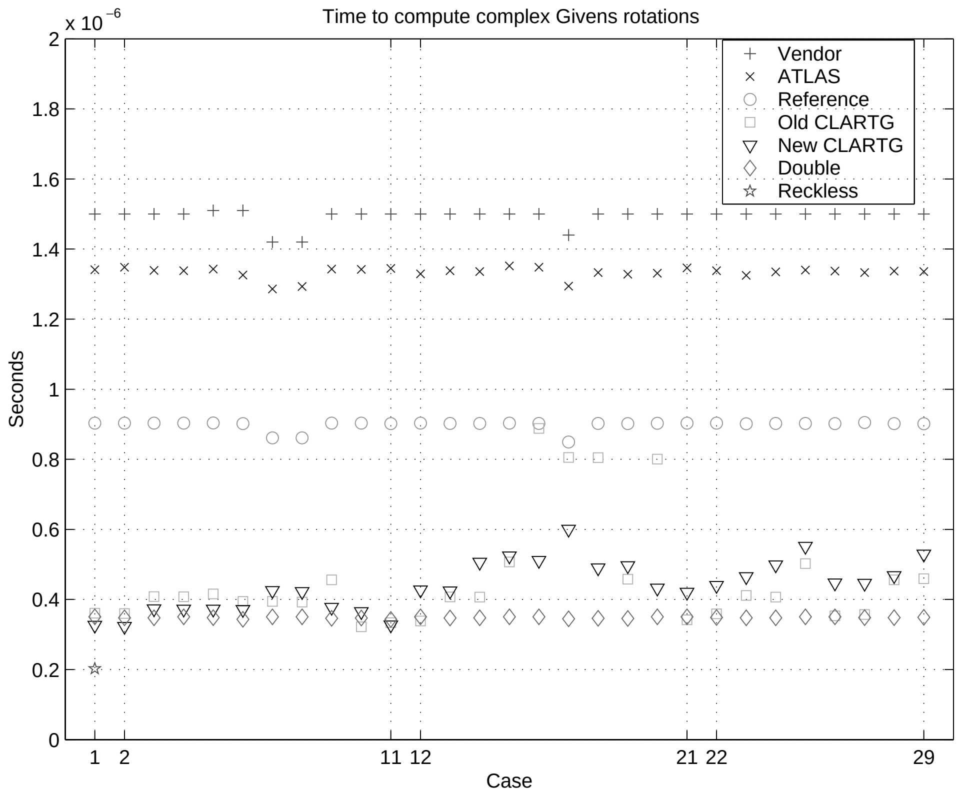

Fig. 5. Time to compute complex Givens rotations.

(6) Simplified new CLARTG in double precision (see below).

(7) "Reckless" CLARTG (see below).

Figure 5 shows absolute times in microseconds, and Figure 6 shows times relative to new CLARTG. The vertical tick marks delimit the cases in the code, as described in Figure 7.

The most common case is Case 1, at the left of the plots. We see that the new CLARTG is a little faster than old CLARTG, and from 2.7 to 4.6 times faster than any version of CROTG.

To get an absolute speed limit, we also ran a "reckless" version of the algorithm that only works in Case 1; that is, it omits all tests for scaling of f and g and simply applies the algorithm appropriate for Case 1. This ultimate version ran in about about 62% of the time of the new CLARTG. This is the price of reliability.

Alternatively, on a system with fast exception handling, one could run this algorithm and then check if an underflow, overflow, or division-by-zero exception occurred, and only recompute in this rare case [Demmel and Li 1994]. This experiment was performed by Doug Priest (private communication) and we report his results here. On a Sun Enterprise 450 server with a 296 MhZ clock, exception handling can be used to (1) save and then clear the floating point

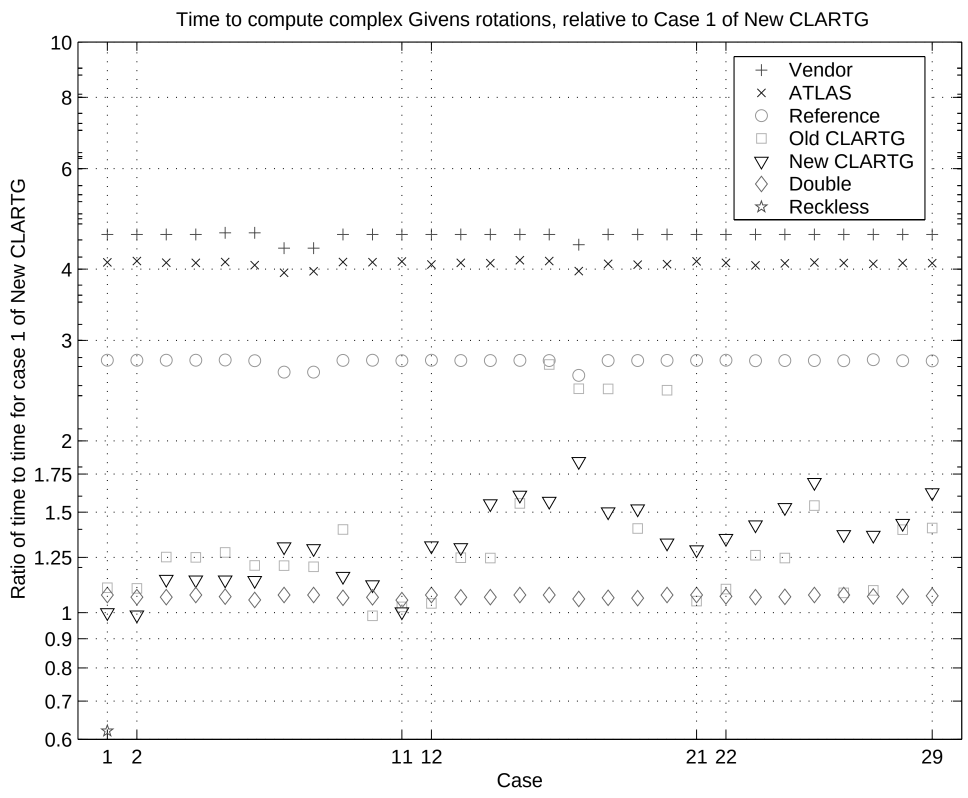

Fig. 6. Relative Time to compute complex Givens rotations.

exceptions on entry to CLARTG, (2) run Case 1 without any argument checking, (3) check exception flags to see if any division-by-zero, overflow, underflow, or invalid operations occurred, (4) use the other cases if there were exceptions, and (5) restore the exception flag on exit. This way arguments falling into the most common usual Case 1 run 25% faster than new CLARTG. Priest notes that it is essential to use in-line assembler to access the exception flags rather than library routines (such as ieee_flags()) which can take 30 to 150 cycles.

Here is a description of the algorithm called "simplified new CLARTG in double precision." It avoids all need to scale and is fastest overall on the above architecture for IEEE single precision inputs: after testing for the cases $f = 0$ or $g = 0$, use Algorithm 3 in Section 6 in IEEE double precision. The three extra exponent bits eliminate over/underflow. On this machine, this algorithm takes about the same time as Case 1 entirely in single. This algorithm is attractive for single precision on this machine, since it is not only fast, but much simpler. Of course, it would not work if the input data were in double, since a wider format is not available on this architecture.

12. COMPUTING REAL GIVENS ROTATIONS

When both f and g are nonzero, the following algorithm minimizes the amount of work:

Case	Case in code	f	g
1	1	(0.11E+01 , 0.22E+01)	(0.33E+01 , 0.44E+01)
2	2	(0.37E+08 , 0.74E+08)	(0.33E+01 , 0.44E+01)
3	2	(0.12E+16 , 0.25E+16)	(0.11E+09 , 0.15E+09)
4	2	(0.42E+23 , 0.83E+23)	(0.37E+16 , 0.50E+16)
5	2	(0.14E+31 , 0.28E+31)	(0.12E+24 , 0.17E+24)
6	2	(0.14E+31 , 0.28E+31)	(0.33E+01 , 0.44E+01)
7	2	(0.14E+31 , 0.28E+31)	(0.26E-29 , 0.35E-29)
8	2	(0.14E+31 , 0.28E+31)	(0.26E-29 , 0.35E-29)
9	2	(0.29E-22 , 0.58E-22)	(0.26E-29 , 0.35E-29)
10	2	(0.98E-15 , 0.20E-14)	(0.87E-22 , 0.12E-21)
11	2	(0.33E-08 , 0.66E-08)	(0.29E-14 , 0.39E-14)
12	3	(0.11E+01 , 0.22E+01)	(0.11E+09 , 0.15E+09)
13	3	(0.37E+08 , 0.74E+08)	(0.37E+16 , 0.50E+16)
14	3	(0.12E+16 , 0.25E+16)	(0.12E+24 , 0.17E+24)
15	3	(0.42E+23 , 0.83E+23)	(0.42E+31 , 0.56E+31)
16	3	(0.11E+01 , 0.22E+01)	(0.42E+31 , 0.56E+31)
17	3	(0.87E-30 , 0.17E-29)	(0.42E+31 , 0.56E+31)
18	3	(0.87E-30 , 0.17E-29)	(0.33E+01 , 0.44E+01)
19	3	(0.87E-30 , 0.17E-29)	(0.87E-22 , 0.12E-21)
20	3	(0.29E-22 , 0.58E-22)	(0.29E-14 , 0.39E-14)
21	3	(0.98E-15 , 0.20E-14)	(0.98E-07 , 0.13E-06)
22	4	(0.37E+08 , 0.74E+08)	(0.11E+09 , 0.15E+09)
23	4	(0.12E+16 , 0.25E+16)	(0.37E+16 , 0.50E+16)
24	4	(0.42E+23 , 0.83E+23)	(0.12E+24 , 0.17E+24)
25	4	(0.14E+31 , 0.28E+31)	(0.42E+31 , 0.56E+31)
26	4	(0.33E-08 , 0.66E-08)	(0.98E-08 , 0.13E-07)
27	4	(0.98E-15 , 0.20E-14)	(0.29E-14 , 0.39E-14)
28	4	(0.29E-22 , 0.58E-22)	(0.87E-22 , 0.12E-21)
29	4	(0.87E-30 , 0.17E-29)	(0.26E-29 , 0.35E-29)

Fig. 7. Input data for timing complex Givens rotations.

Algorithm 9: Real Givens rotations when *f* and *g* are nonzero, without scaling

```
FG2 = F**2 + G**2
R   = sqrt(FG2)
RR  = 1/R
C   = abs(F)*RR
S   = G*RR
if F < 0 then
    S = -S
    R = -R
endif
```

We may now apply the same kind of analysis that we applied to Algorithm 3. We just summarize the results here.

Our ultimate algorithm below does one division and one square root. It is nearly identical to the SLARTG from LAPACK 3.0, differing only in (1) choosing the sign bits of the returned quantities, (2) care in scaling to avoid infinite loops when an input is infinite, and (3) multiplying by RR twice instead of dividing by R twice. In contrast, the SROTG routine in the Fortran Reference BLAS does one square root and two to four divisions to compute the same quantities (this depends on how aggressively the compiler optimizes the code). It contains 96 noncomment lines of code, as opposed to 22 lines for the reference BLAS SROTG (20 lines excluding 2 described below), and is contained in Bindel et al. [2002] and Blackford et al. [2001].

Algorithm 10: Real Givens rotations when *f* and *g* are nonzero, with scaling

```
scale = max( abs(F) , abs(G) )
if scale > $z^2$ then
    scale F, G and scale down by powers of $z^{-2}$ until scale $\le z^2$
elseif scale < $z^{-2}$ then
    scale F, G and scale up by powers of $z^2$ until scale $\ge z^{-2}$
endif
FG2 = F**2 + G**2
R   = sqrt(FG2)
RR  = 1/R
C   = abs(F)*RR
S   = G*RR
if F < 0 then
    S = -S
    R = -R
endif
unscale R if necessary
```

13. ACCURACY RESULTS FOR REAL GIVENS ROTATIONS

The accuracy of a variety of routines was measured in a way entirely analogous to the way described in Section 10. The results are shown in the tables below.

First, consider the results in the absence of underflow. All three versions of SROTG use a scale factor $|f|+|g|$ that can overflow even when r does not, or not overflow but its reciprocal underflow (its reciprocal is not explicitly computed, but may be after aggressive compiler optimizations). Eliminating these extreme values of f and g from the tests yields the results in the lines labeled "Limited."

With gradual underflow, letting f and g both equal the smallest positive denormalized number m yields $s = c = 1$ instead of $1/\sqrt{2}$, a very large relative error. This is because $r = m$ is the best machine approximation to the true result $\sqrt{2}m$, after which $f = m$ and $g = m$ are divided by m to get c and s, respectively. Slightly larger f and g yield slightly smaller (but still quite large) relative errors in s and c.

Without Gradual Underflow			
Routine	Max error in r_s	Max error in s_s	Max error in c_s
New SLARTG	1.19	2.20	2.20
Old SLARTG	1.19	2.20	2.20
Reference SROTG	NaN	NaN	NaN
Limited Reference SROTG	1.94	2.69	2.69
ATLAS SROTG	NaN	NaN	NaN
Limited ATLAS SROTG	1.93	2.20	2.20
Vendor SROTG	NaN	NaN	NaN
Limited Vendor SROTG	1.94	2.26	2.26

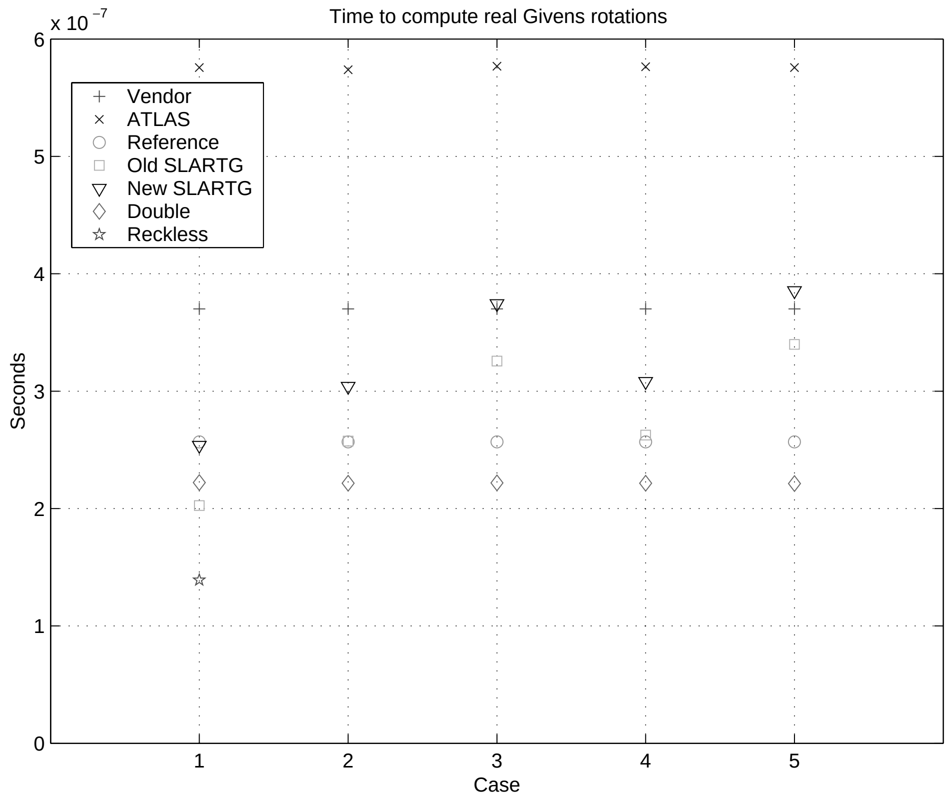

Fig. 8. Time to compute real Givens rotations.

With Gradual Underflow			
Routine	Max error in r_s	Max error in s_s	Max error in c_s
New SLARTG	1.19	2.20	2.20
Old SLARTG	1.19	2.20	2.20
Reference SROTG	NaN	NaN	NaN
Limited Reference SROTG	1.93	$7 \cdot 10^6$	$7 \cdot 10^6$

14. TIMING RESULTS FOR REAL GIVENS ROTATIONS

Seven routines to compute real Givens rotations were tested in a way entirely analogous to the manner described in Section 11. The test arguments and timing results are shown in the table and in Figures 8 and 9.

Even though the new and old SLARTGs are essentially identical (especially for Case 1, when no scaling is required), the new SLARTG runs about 20% slower than the old one. This time difference is quite repeatable (and does not depend on whether we multiply twice by RR or divide twice by R.). In contrast, with an earlier version of the compiler the times were roughly reversed, with our new SLARTG fastest. The exact performance is clearly sensitive to changes in the compiler.

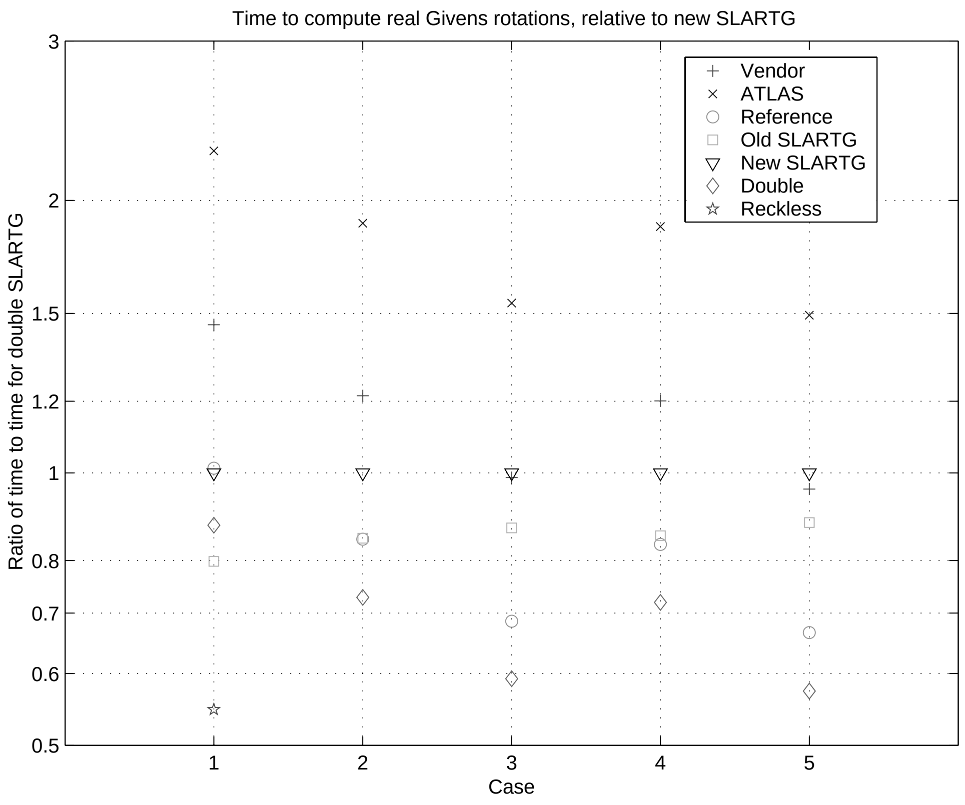

Fig. 9. Relative Time to compute real Givens rotations.

All three versions of SROTG (reference, ATLAS, and Sun's vendor version) originally computed more than just s, c and r: they compute a single scalar z from which one can reconstruct both s and c. It is defined by

$$z = \begin{cases} s & \text{if } |f| > |g| \\ 1/c & \text{if } |f| \le |g| \text{ and } c \neq 0 \\ 1 & \text{otherwise} \end{cases}$$

The three cases can be distinguished by examining the value of z and then s and c reconstructed. This permits, for example, the QR factors of a matrix A to overwrite A when Givens rotations are used to compute Q, as is the case with Householder transformations. This capability is not used in LAPACK, so neither version of SLARTG computes z. To make the timing comparisons fairer, we therefore removed the two lines of code computing z from the reference SROTG when doing the timing tests below. We did not however modify ATLAS or the Sun performance library in any way, so those routines do slightly more work than necessary.

Input data for timing real Givens rotations		
Case	f	g
1	0.11E+01	0.33E+01
2	0.12E+16	0.37E+16
3	0.14E+31	0.42E+31
4	0.98E−15	0.29E−14
5	0.87E−30	0.26E−29

To get an absolute speed limit, we also ran a version of the algorithm that only works in Case 1; that is, it omits all tests for scaling of f and g and simply applies the algorithm appropriate for data that is not too large or too small. This ultimate version ran in about 55% of the time of the new SLARTG. This is the price of reliability.

Experiments by Doug Priest using exception handling to avoid branching showed an 8% improvement in the most common case, when no scaling was needed (using f77 version SUNWspro5.0).

Finally, the double precision version of SLARTG simply tests for the cases $f = 0$ and $g = 0$, and then runs Algorithm 9 in double precision without any scaling. It is fast and simple.

15. CONCLUSIONS

We have justified the specification of Givens rotations put forth in the recent BLAS Technical Forum standard. We have shown how to implement the new specification in a way that is both faster than previous implementations, and more reliable. We used a systematic design process for such kernels that could be used whenever accuracy, reliability against over/underflow, and efficiency are simultaneously desired. A side effect of our approach is that the algorithms can be much longer than before when they must be implemented in the same precision as the arguments, but if fast arithmetic with wider range is available to avoid over/underflow, the algorithm becomes very simple, just as reliable, and at least as fast.

ACKNOWLEDGMENTS

The authors would like to acknowledge Ed Anderson for extensive comments that helped improve this article, and Doug Priest for performing performance experiments and letting us report them here.

REFERENCES

Anderson, E. 2000. Discontinuous plane rotations and the symmetric eigenvalue problem. Computer Science Dept. Tech. Rep. CS-00-454. Univ. Tennessee, Knoxville, Tenn. www.netlib.org/lapack/lawns/lawn150.ps.

Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., Blackford, S., and Sorensen, D. 1999. *LAPACK Users' Guide 3rd ed.* SIAM, Philadelphia, Pa.

Anderson, E. and Fahey, M. 1997. Performance improvements to LAPACK for the Cray scientific library. Computer Science Dept. Tech. Rep. CS-97-359. Univ. Tennessee, Knoxville, Tenn. www.netlib.org/lapack/lawns/lawn126.ps.

ANSI/IEEE Std 754-1985, *IEEE Standard for Binary Floating Point Arithmetic*. ANSI/IEEE, New York.

BINDEL, D., DEMMEL, J., KAHAN, W., AND MARQUES, O. 2002. Software for reliable and efficient Givens rotations. `www.cs.berkeley.edu/~demmel/Givens`.

BLACKFORD, S., CORLISS, G., DEMMEL, J., DONGARRA, J., DUFF, I., HAMMARLING, S., HENRY, G., HEROUX, M., HU, C., KAHAN, W., KAUFMAN, L., KEARFOTT, B., KROGH, F., LI, X., MAANY, Z., PETITET, A., POZO, R., REMINGTON, K., WALSTER, W., WHALEY, C., WOLFF V. GUDENBERG, J., AND LUMSDAINE, A. 2001. Basic Linear Algebra Subprograms Technical (BLAST) Forum Standard. *Intern. J. High Performance Comput. 15*, 3–4 (also available at `www.netlib.org/blas/blast-forum/`).

DEMMEL, J. 1997. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, Pa.

DEMMEL, J. AND LI, X. 1994. Faster numerical algorithms via exception handling. *IEEE Trans. Comput. 43*, 8, 983–992. LAPACK Working Note 59.

DONGARRA, J., BUNCH, J., MOLER, C., AND STEWART, G. W. 1979. *LINPACK User's Guide*. SIAM, Philadelphia, Pa.

GOLUB, G. AND VAN LOAN, C. 1996. *Matrix Computations*, 3rd ed. Johns Hopkins University Press, Baltimore, Md.

HULL, T., FAIRGRIEVE, T., AND TANG, P. T. P. 1994. Implementing complex elementary functions using exception handling. *ACM Trans. Math. Softw. 20*, 2 (June), 215–244.

HULL, T., FAIRGRIEVE, T., AND TANG, P. T. P. 1997. Implementing complex Arcsine and Arccosine using exception handling. *ACM Trans. Math. Softw. 23*, 3 (Sept), 299–335.

IEEE 754-2002. *IEEE Standard for Binary Floating Point Arithmetic Revision.* `grouper.ieee.org/groups/754`.

KATO, T. 1980. *Perturbation Theory for Linear Operators*, 2 ed. Springer-Verlag, Berlin, Germany.

KUNKEL, P. AND MEHRMANN, V. 1991. Smooth factorizations of matrix valued functions and their derivatives. *Num. Math. 60*, 115–132.

LAWSON, C., HANSON, R., KINCAID, D., AND KROGH, F. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Softw. 5*, 308–323.

SMITH, B. T., BOYLE, J. M., DONGARRA, J. J., GARBOW, B. S., IKEBE, Y., KLEMA, V. C., AND MOLER, C. B. 1976. *Matrix Eigensystem Routines—EISPACK Guide*. Lecture Notes in Computer Science, vol. 6. Springer-Verlag, Berlin, Germany.

STEWART, G. W. 1976. The economical storage of plane rotations. *Numer. Math. 25*, 2, 137–139.

WILKINSON, J. H. 1965. *The Algebraic Eigenvalue Problem*. Oxford University Press, Oxford, England.

Received February 2001; revised January 2002; accepted February 2002