On computing Givens rotations reliably and efficiently

D. Bindel* J. Demmel† W. Kahan‡ O. Marques§

January 31, 2001

Abstract

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 25% faster than the previous LAPACK implementation in the most common case, and nearly 4 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.

1 Introduction

Givens rotations 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

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

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

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

*Computer Science Division University of California, Berkeley, CA 94720 (dbindel@cs.berkeley.edu). This material is based upon work supported under a National Science Foundation Graduate Research Fellowship.

†Computer Science Division and Mathematics Dept., University of California, Berkeley, CA 94720 (demmel@cs.berkeley.edu). This material is based in part upon work supported by the 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.

‡Computer Science Division and Mathematics Dept., University of California, Berkeley, CA 94720 (wkahan@cs.berkeley.edu).

§NERSC, Lawrence Berkeley National Lab, (osni@nersc.gov).

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

From this we see that

$$|c|^2 + |s|^2 = 1 \quad \text{and} \quad c - \bar{c} = 0, \ \text{ i.e. } c \text{ is real} \tag{2}$$

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

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

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 [1] routines SLARTG, CLARTG, SLARGV and CLARGV, the Level 1 BLAS routines SROTG and CROTG [6], as well as Algorithm 5.1.5 in [5] 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 [3], 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 rotation 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 5 square roots and 9 divisions, and perhaps 13 divisions, depending on the implementation of the complex absolute value function cabs.

However, these 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 $\|f\| = \max(|\mathtt{re}\, f|, |\mathtt{im}\, f|)$. 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 (nonconvex) 2D polygonal region S (for 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 25% faster than LAPACK's CLARTG routine, and nearly 4 times faster than the CROTG routine in the vendor BLAS on a Sun Ultra-10, ATLAS BLAS, or Fortran reference BLAS.

If $(\log \|f\|, \log \|g\|)$ lies outside S, there are two possibilities: scaling f and g by a constant to fit inside S, or using different formulas. Scaling may be interpreted geometrically as shifting S parallel to the diagonal line $\log \|f\| = \log \|g\|$ in $(\log \|f\|, \log \|g\|)$ space. The region covered by shifted images of S (S'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 S and then use the previous formula.

The remaining region of $(\log \|f\|, \log \|g\|)$ space, including space outside S'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 $|g|^2$ (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 4 cases, there are 2 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, i.e. 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 [4]). 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 4 cases is the best we know.

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 nearly fully accurate.

The rest of this paper 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 compares the accuracy of our new complex Givens routine and several alternatives; only ours is accurate in all cases. Section 10 discusses the performance of our complex Givens routine. Sections 11, 12 and 13 discuss algorithms, accuracy and timing for real Givens rotations, which are rather easier. Section 14 draws conclusions. The actual software is included in an appendix.

2 Defining Givens rotations

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

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

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

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

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 when at least one of f and g are nonzero, the most that we can deduce from the first component of $R(c, s)[f, g]^T = [r, 0]^T$ in (1) is that

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

for $i = \sqrt{-1}$ and some real θ. From the fact that c must be real we deduce that if $f \neq 0$ then

$$\begin{aligned} c &= \pm \frac{|f|}{\sqrt{|f|^2 + |g|^2}} \\ s &= \pm \mathrm{sign}(f) \frac{\bar{g}}{\sqrt{|f|^2 + |g|^2}} \\ r &= \pm \mathrm{sign}(f) \sqrt{|f|^2 + |g|^2} \end{aligned} \tag{3}$$

and if $f = 0$ and $g \neq 0$ then

$$\begin{aligned} c &= 0 \\ s &= e^{i\theta} \\ r &= e^{i\theta} g \end{aligned} \tag{4}$$

As stated before, when $f = g = 0$, c and s can be chosen arbitrarily, as long as they satisfy (2).

The extra requirements initially chosen by the BLAS Technical Forum to help resolve the choice of $\pm$ sign in (3) and θ in (4) are as follows.

R1 The definitions for real and complex data should be consistent, so that real data passed to the complex algorithm should result in the same answers (modulo roundoff) as from the real algorithm.

R2 Current LAPACK subroutines that use Givens rotations should continue to work correctly with the new definition.

The current LAPACK subroutines SLARTG and CLARTG (which compute a single real and complex Givens rotation, resp.) do not satisfy requirement 1. Furthermore, the LAPACK subroutines SLARGV and CLARGV for computing multiple Givens rotations do not compute the same answers as SLARTG and CLARTG, resp. The differences are described in section 3 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.)

However, R1 and R2 do not immediately resolve the choice of sign in (1). To proceed we add requirement

R3 The mapping from (f, g) to (c, s, r) should be continuous whenever possible.

Continuity of c and s as functions of f and g is not possible everywhere, because as real f and g approach $(0, 0)$ along the real line $g = f \cdot \tan\alpha$, $c = \pm\cos\alpha$ and $\pm s = \sin\alpha$, so c and s must be discontinuous at $(0, 0)$.

But consider c, s, r as functions of $(f, g) = (e^{i\alpha}, 1)$ as α increases from 0 to 2π, i.e. f traverses the unit circle in the complex plane. At $\alpha = 0$, $(f, g) = (1, 1)$ and consider the common convention $(c, s) = (\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}})$. As α increases, $|c| = |s|$ remains equal to $\frac{1}{\sqrt{2}}$. Since c is real, continuity implies c stays fixed at $c = \frac{1}{\sqrt{2}}$ for all α, and hence $s = e^{i\alpha}/\sqrt{2}$ and $r = e^{i\alpha}\sqrt{2}$ are continuous as desired. Thus requirement R3 implies that c must be nonnegative. Together with (3), this implies that when $f \neq 0$ we have

$$\begin{aligned} c &\equiv \frac{|f|}{\sqrt{|f|^2 + |g|^2}} \\ s &\equiv \mathrm{sign}(f) \frac{\bar{g}}{\sqrt{|f|^2 + |g|^2}} \\ r &\equiv \mathrm{sign}(f) \sqrt{|f|^2 + |g|^2} \end{aligned} \tag{5}$$

Formulas (5) obviously define f, g and r continuously away from $f = 0$. When $g = 0$, they simplify to $c = 1$, $s = 0$ and $r = f$. This is attractive because $R(1, 0)$ is the identity matrix, so using it to multiply an arbitrary pair of vectors requires no work,

When $f = 0$ but $g \neq 0$ we reexamine (4) in the light of requirement R3. Since c and s are not continuous at $f = 0$, because $\mathrm{sign}(f)$ can change arbitrarily in a small complex neighborhood of 0, we cannot hope to define θ by a continuity argument that includes complex f. Instead, we ask just that c, s, and r be

continuous functions of real $f \geq 0$ and complex $g \neq 0$, i.e. they should be continuous as f approaches zero from the right. This limit is easily seen to be

$$\begin{aligned} c &\equiv 0 \\ s &\equiv \mathrm{sign}(\bar{g}) \\ r &\equiv |g| \end{aligned} \tag{6}$$

which we take as the definition for $f = 0$ and complex $g \neq 0$.

Finally we consider the case $f = g = 0$. This is impossible to define by continuity, since f and g can approach 0 from any direction, so instead we add requirement

R4 Given a choice of c and s, choose those requiring the least work.

Since $R(c, s)$ is typically used to multiply a pair of vectors, and $R(1, 0) = I$ requires no work to do this, we set $c = 1$ and $s = 0$ when $f = g = 0$.

In summary, the algorithm for complex or real f and g is as follows.

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

When f and g are real, the algorithm can be slightly simplified by replacing $\bar{g}$ by g.

2.1 Exceptional cases

When this algorithm is run in IEEE floating point arithmetic [2] 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 insist that the routine must terminate and return some output values in all cases.

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 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 $s = 0$, since these require no arithmetic operations to compute. Rather than specify exactly what should happen when an input is NaN, we insist only that *at least* r be a NaN, and perhaps c and s as well, 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.

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 implemented (in hardware or software) as "if $(a > b)$ then a else b" then $\max(0, NaN)$ returns NaN but $\max(NaN, 0)$ returns 0. On the other hand, the equally reasonable implementation "if $(a < b)$ then b else a" instead returns 0 and NaN, respectively. 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. 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, 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 $c = 0$, $s = NaN$ and $r = \infty$ if $f = 0$ and $g = \infty + i \cdot \infty$ since $s = \mathrm{sign}(\bar{g}) = \bar{g}/g$ cannot be well-defined while $r = |g|$ can be. Or one could simply return NaNs even if a limit existed, for example returning $c = 0$, $s = NaN$ and $r = NaN$ 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 implementor's discretion which approach to take. But we insist that at least r either be infinite or a NaN.

The assiduous reader will have noted that Algorithm 1 leaves ambiguous how the sign of zero is treated (IEEE arithmetic includes both $+0$ and -0). Different implementations are free to return $+0$ or -0 whenever a zero is to be delivered. 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 [1] and earlier versions, and in the Level 1 BLAS [6]. The LAPACK algorithms in question are SLARTG, CLARTG, SLARGV and CLARGV, and the Level 1 BLAS routines are SROTG and CROTG. 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 requirement R2.

SLARTG When $f = 0$ and $g \neq 0$, Algorithm 1 returns $s = \mathrm{sign}(g)$ 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 9 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 $c = \mathrm{sign}(f)$ instead of $c = 1$. When $f = 0$ and $g \neq 0$, SLARGV returns $s = 1$ instead of $s = \mathrm{sign}(g)$. When $f \neq 0$ and $g \neq 0$, SLARGV returns $\mathrm{sign}(c) = \mathrm{sign}(f)$, instead of $c \geq 0$.

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

SROTG SROTG overwrites f by r and g by a quantity z from which one can reconstruct both s and c ($z = s$ 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 that 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 $s = \mathrm{sign}(\bar{g})$ 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 [2], 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 `SAFMIN` may be set 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 the effective underflow threshold is $\texttt{SAFMIN}/\varepsilon$, since underflow in x can cause a relative error in $\texttt{SAFMIN}/\varepsilon + x$ of at most ε, the same as roundoff.

In our scaling algorithms we will use the quantity $z = (\varepsilon/\texttt{SAFMIN})^{1/4}$ rounded to the nearest power of the radix. Thus we use $z^{-4} = \texttt{SAFMIN}/\varepsilon$ 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. 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 [2] are used.

In later discussion we denote the actual overflow threshold by OV, the underflow threshold by UN, and the smallest positive number by m, which is $2 \cdot \varepsilon \cdot \text{UN}$ 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 $\|w\| = \max(|\texttt{re}\, w|, |\texttt{im}\, w|)$ 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^±4 so that z^-2 ≤ ||G|| ≤ 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^±4
else
      ... both F and G are nonzero
      ... use algorithm described below
endif
```

We note that even though $\texttt{F} = 0 \neq \texttt{G}$ 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 2.1, 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

```
SCALEG = ||G||
SCALEF = ||F||
if SCALEG = 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 SCALEF = 0
      ... G must be nonzero
      C = 0
      scale G by powers of z^±4 so that z^-2 ≤ ||G|| ≤ 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^±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 SCALEG=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 SCALEF = 0 can be taken when G is a NaN or infinity. This means that a loop to scale G (and SCALEG) 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	$\frac{f}{\|f\|}\frac{\overline{g}}{\sqrt{\|f\|^2+\|g\|^2}}$

Now recall $z = (\varepsilon/\mathtt{SAFMIN})^{1/4}$, 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 |R| roughly between z^{-1} and z^2. If the smaller component of R underflows, it is deserved.

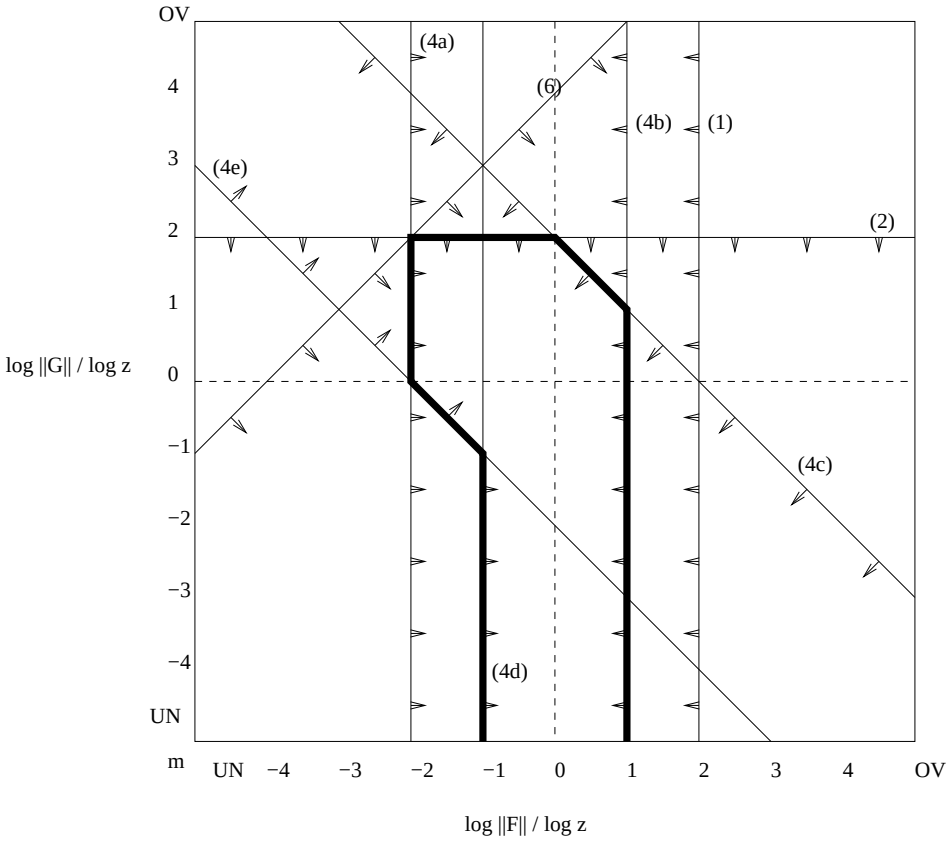

Figure 1: Inequalities describing the region of no unnecessary over/underflow. UN and OV are the over/underflow thresholds; m is the smallest representable positive number.

8. This line is safe given previous assumptions, returning $|\mathtt{S}|$ 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.

Note that 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 S. The boundary of S is complicated, so the time to test for membership in S 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 $\|g\|$. So we test for membership in the subset of S labeled (1) in Figure 2 by the following algorithm:

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 $z^{\pm 1} = 2^{\pm 25} \approx 10^{\pm 7}$ and in single between $z^{\pm 1} = 2^{\pm 242} \approx 10^{\pm 73}$ in double), which we expect to be most arguments, especially in double.

The complement of Region (1) in S 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 the case, but do something else instead.

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 if 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 2.1. 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

$$\begin{aligned} c &\approx 1 \\ s &\approx \mathrm{sign}(f)\frac{\bar{g}}{|f|} = \frac{f \cdot \bar{g}}{|f|^2} \\ r &\approx f \end{aligned} \tag{7}$$

This region is closely approximated by the regions $\|g\| \leq \varepsilon^{1/2}\|f\|$ 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

$$\begin{aligned} c &\approx \frac{|f|}{|g|} = \frac{|f|^2}{|f| \cdot |g|} \\ s &\approx \mathrm{sign}(f)\frac{\bar{g}}{|g|} = \frac{f \cdot \bar{g}}{|f| \cdot |g|} \\ r &\approx \mathrm{sign}(f)|g| = \frac{f \cdot |g|^2}{|f| \cdot |g|} \end{aligned} \tag{8}$$

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

An important difference between the formulas in (7) and (8) versus the formula (5) is that (7) and (8) 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 (5) f and g must be scaled by the same value.

Here are the algorithms implementing (7) and (8) without scaling. Note that (7) does not even require a square root.

Algorithm 4: Computing complex Givens rotations when $\|g\| \leq \sqrt{\varepsilon}\|f\|$, using formulas (7), without scaling

```
if ||G|| < sqrt(ε) · ||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 $\|f\| \leq \sqrt{\varepsilon}\|g\|$, using formulas (8), without scaling

```
if ||F|| < sqrt(ε) · ||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 S, 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 S in Figure 2) where membership can be easily tested. This safe region for Algorithm 4 is the set satisfying

$$z^{-2} \leq \|f\| \leq z^2 \quad \text{and} \quad z^{-2} \leq \|g\| \leq z^2 \tag{9}$$

This safe region for Algorithm 5 is the smaller set satisfying

$$z^{-1} \leq \|f\| \leq z \quad \text{and} \quad z^{-1} \leq \|g\| \leq z \tag{10}$$

This leads to the following algorithms, which incorporate scaling.

Algorithm 6: Computing complex Givens rotations when $\|g\| \leq \sqrt{\varepsilon}\|f\|$, using formulas (7), with scaling

```
if ||G|| < sqrt(ε) · ||F|| then
    C = 1
    R = F
    scale F by powers of z^±4 so z^-2 ≤ ||F|| ≤ z^2
    scale G by powers of z^±4 so z^-2 ≤ ||G|| ≤ z^2
    D1 = 1/(re(F)**2 + im(F)**2)
    S = F * conj(G)
    S = S * D1
    unscale S by powers of z^±4 to undo scaling of F and G
end if
```

Algorithm 7: Computing complex Givens rotations when $\|f\| \leq \sqrt{\varepsilon}\|g\|$, using formulas (8), with scaling

```
if ||F|| < sqrt(ε) · ||G|| then
    scale F by powers of z^±2 so z^-1 ≤ ||F|| ≤ z
    scale G by powers of z^±2 so z^-1 ≤ ||G|| ≤ 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^±2 to undo scaling of F and G
endif
```

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

7.1 Exceptional cases

First consider Case 2, i.e. Algorithm 6. It is possible for either `F` or `G` to be NaNs (since $\|\mathtt{F}\|$ and $\|\mathtt{G}\|$ may not be) but if neither is a NaN then only `F` can be infinite (since the test is $\|\mathtt{G}\| < \sqrt{\varepsilon} \cdot \|\mathtt{F}\|$, not $\|\mathtt{G}\| \leq \sqrt{\varepsilon} \cdot \|\mathtt{F}\|$). 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` independently of whether inputs are infinite or NaNs. `S` is a NaN if either `F` or `G` is a NaN or infinite. If we simply get `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, i.e. 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 S. It is geometrically apparent that the set of points scalable in regions (4a) and (4b)of Figure 2 lie in the set of all diagonal translates of S, i.e. the "shadow" of S, and can be scaled to lie in S. Indeed, all points in region (2) and many (but not all) points in region (3) can be scaled to lie in S, 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, i.e. (f, g) is in A1, we multiply f and g by z^{-1}, putting it into A2. 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 like in the union of the two parallelograms B1 and B2 in Figure 4. Then, if s

is still less than z^{-1}, i.e. (f, g) is in B1, we multiply f and g by z, putting it 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 $\|\mathtt{F}\| > 1$
    scale `F` and `G` down by powers of z^{-2} until $\max(\|\mathtt{F}\|, \|\mathtt{G}\|) \leq z^2$
    if $\max(\|\mathtt{F}\|, \|\mathtt{G}\|) > z$, scale `F` and `G` down by z^{-1}
else
    scale `F` and `G` up by powers of z^2 until $\|\mathtt{F}\| \geq z^{-2}$
    if $\|\mathtt{F}\| < z^{-1}$, 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`

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 in included in the Appendix. It contains 248 noncomment lines, as opposed to 20 in the reference CROTG implementation.

8.1 Exceptional cases

Either input may be a NaN, and they may be simultaneously infinite. In any of these cases, all three outputs will be NaNs. As before, care must be taken in scaling.

9 Accuracy results for complex Givens rotations

The algorithm was run for $46^4 = 4477456$ values of f and g, where the real and imaginary part of f and g independently took on 46 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. The correct answer inputs 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

$$|r_s - r_d| / \max(\varepsilon |r_d|, \mathtt{SAFMIN}) \tag{11}$$

and with gradual underflow it is

$$|r_s - r_d| / \max(\varepsilon |r_d|, \mathtt{SAFMIN} * 2 * \varepsilon) \tag{12}$$

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 $\mathtt{SAFMIN} * 2 * \varepsilon$ 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 with the -fast -O5 flags, which means gradual underflow is *not* used, i.e. results less than `SAFMIN` are replaced by 0. Therefore we expect the measure (11)

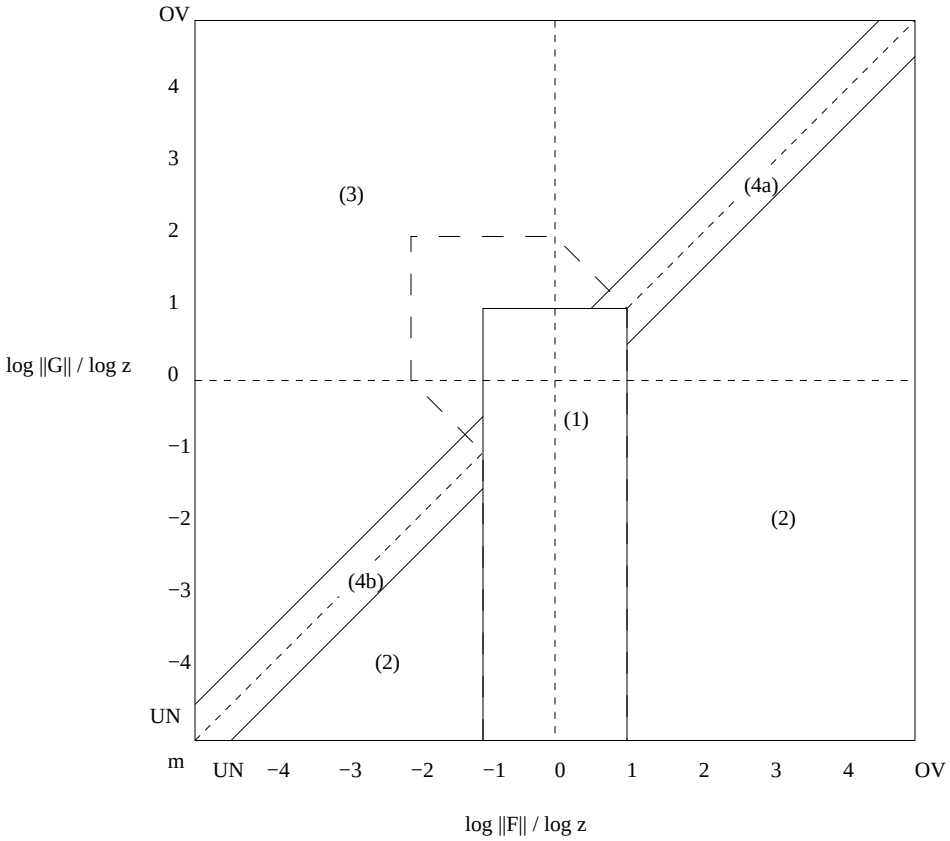

Figure 2: Cases in the code when $f \neq 0$ and $g \neq 0$

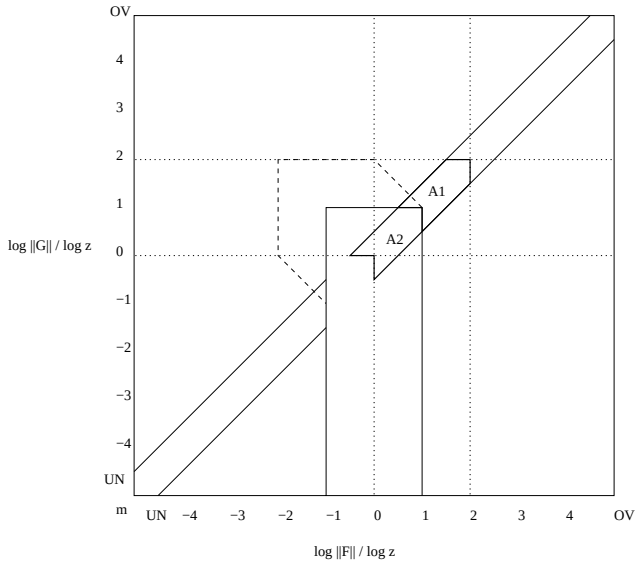

Figure 3: Scaling when (f, g) is in Region (4a).

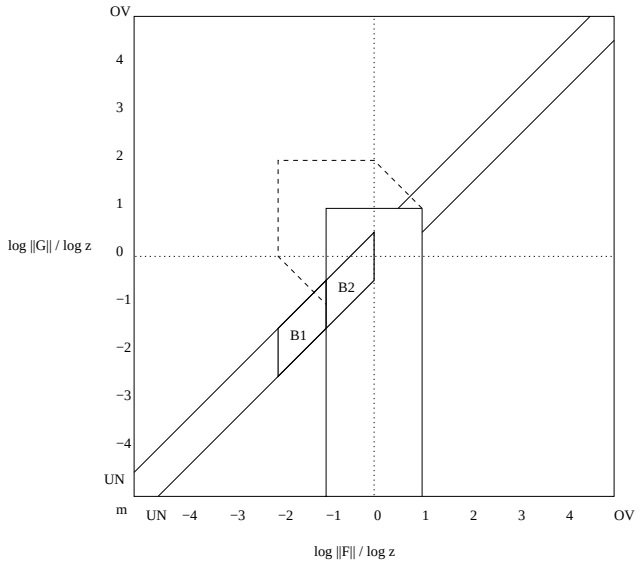

Figure 4: Scaling when (f, g) is in Region (4b).

to be at least 1, 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 gradual underflow is used, so we expect the more stringent measure (12) to be close to 1.

The results are as follows:

Without Gradual Underflow			
Routine	Max error in r_s	Max error in s_s	Max error in c_s
New CLARTG	3.04	2.96	2.56
Old CLARTG	70588	70588	70292
Reference CROTG	NaN	NaN	NaN
Modified Reference CROTG	3.59	3.41	3.22
ATLAS CROTG	NaN	NaN	NaN
Limited ATLAS CROTG	2.88	$2 \cdot 10^7$	3.11
Vendor CROTG	NaN	NaN	NaN
Limited Vendor CROTG	3.59	$2 \cdot 10^7$	3.22

With Gradual Underflow			
Routine	Max error in r_s	Max error in s_s	Max error in c_s
New CLARTG	3.04	2.96	3.04
Old CLARTG	4.60	4.27	4913930
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. The sine is computed as $s = (\frac{f}{|f|}) \cdot (\bar{g})/(\sqrt{|f|^2 + |g|^2})$, where the multiplication is done first. All three quantities in parentheses are 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 .4. This can be repaired by inserting parentheses $s = (\frac{f}{|f|}) \cdot ((\bar{g})/(\sqrt{|f|^2 + |g|^2}))$ so the division is done first. Excluding these cases where $|f| + |g|$ overflows, 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 and vendor version 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. 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.

10 Timing results for complex Givens rotations

For complex Givens rotations, we compared the new algorithm described above, the old CLARTG from LAPACK, and CROTG from the reference BLAS. Timings were done on a Sun Ultra-10 using the f77 compiler 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. 29 cases were tried in all, exercising all paths in the new CLARTG code. The input data is 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 ot 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. Six 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
6. Simplied new CLARTG in double precision (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 the table below.

The most common case is Case 1, at the left of the plots. We see that the new CLARTG is about 25% faster than old CLARTG, and nearly 4 times faster than any version of CROTG.

To get an absolute speed limit, we also ran a version of the algorithm that only works in Case 1; i.e. 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 .243 microseconds, about 68% 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 [4]. This experiment was performed by Doug Priest [7] 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 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 IEEE double precision. The three extra exponent bits eliminate over/underflow. On this machine, this algorithm takes about .365 microseconds for all nonzero inputs f and g, nearly exactly the same 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.

Input data for timing complex Givens rotations			
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)

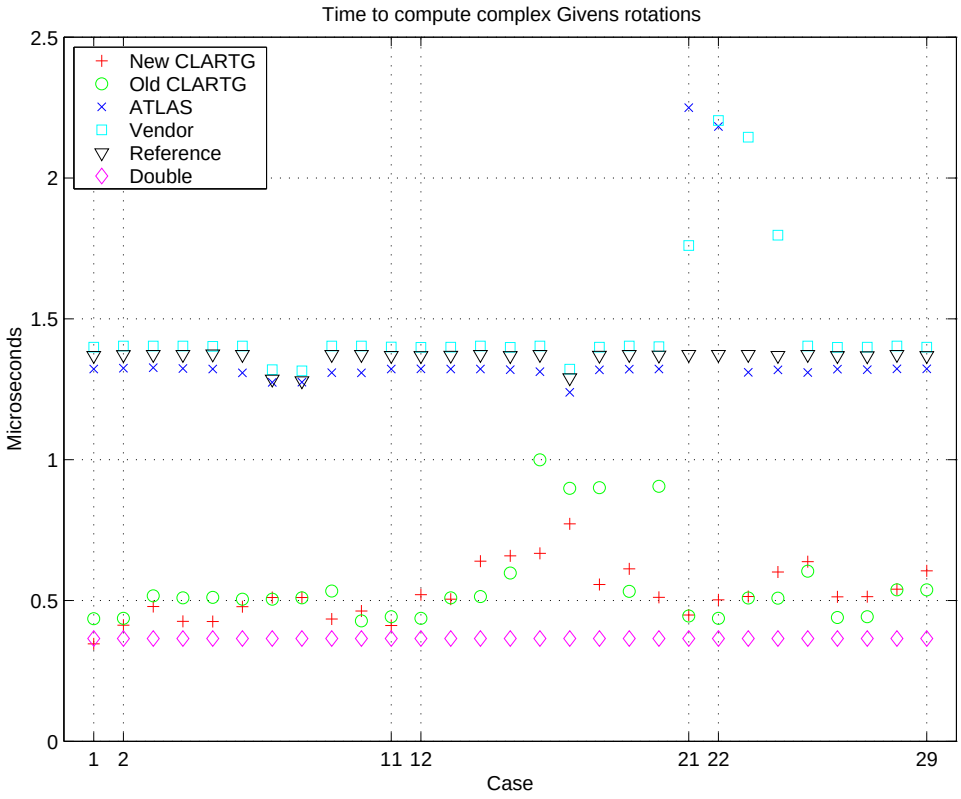

Figure 5: Time to compute complex Givens rotations.

11 Computing real Givens rotations

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

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.

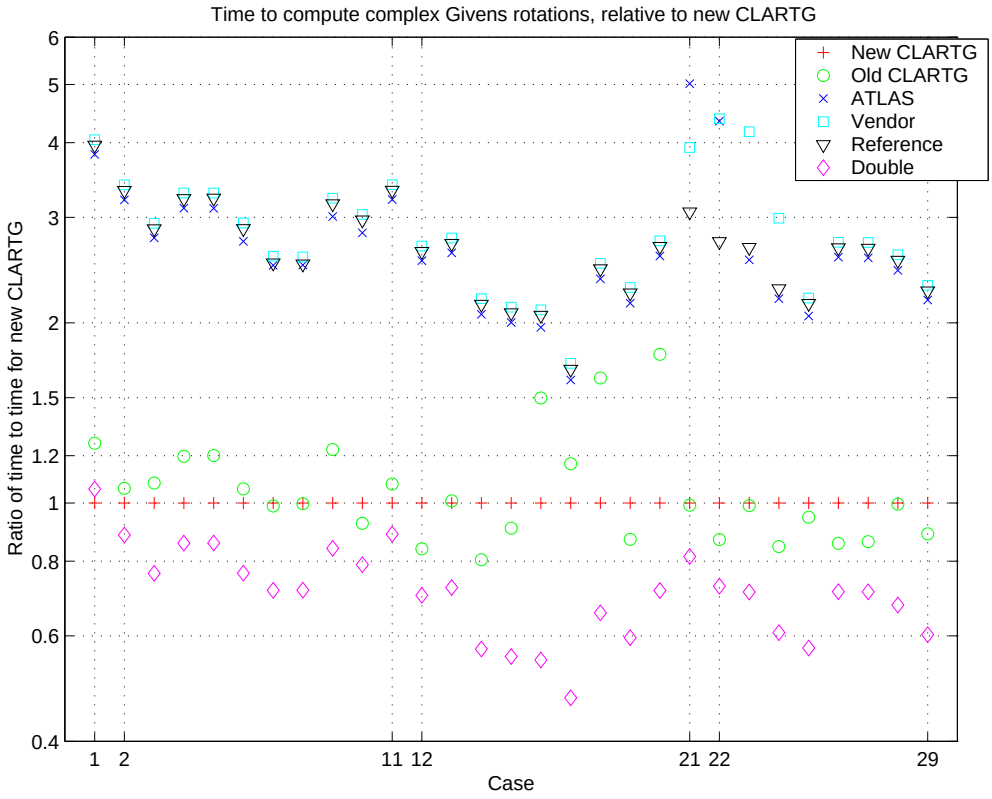

Figure 6: Relative Time to compute complex Givens rotations.

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` $\leq z^2$
elseif `scale` $< z^{-2}$ then
    scale `F`, `G` and `scale` up by powers of z^2 until `scale` $\geq z^{-2}$
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

This algorithm does one division and one square root. In contrast, the SROTG routine in the Fortran Reference BLAS does 1 square root and 4 divisions to compute the same quantities. It contains 95 noncomment lines of code, as opposed to 22 lines for the reference BLAS SROTG (20 lines excluding 2 described below), and is contained in the appendix.

12 Accuracy results for real Givens rotations

The accuracy of a variety of routines were measured in a way entirely analogous to the way described in section 9. The results are shown in the tables below.

First consider the results in the absence of of gradual underflow. All three versions of SROTG use a scale factor $|f| + |g|$ which can overflow even when r does not. 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	1.45	1.81	1.81
Old SLARTG	1.45	1.81	1.81
Reference SROTG	NaN	NaN	NaN
Limited Reference SROTG	1.51	1.95	1.95
ATLAS SROTG	NaN	NaN	NaN
Limited ATLAS SROTG	1.68	1.55	1.55
Vendor SROTG	NaN	NaN	NaN
Limited Vendor SROTG	1.68	1.55	1.55

With Gradual Underflow			
Routine	Max error in r_s	Max error in s_s	Max error in c_s
New SLARTG	1.45	1.81	1.81
Old SLARTG	1.45	1.81	1.81
Reference SROTG	NaN	NaN	NaN
Limited Reference SROTG	1.51	$7 \cdot 10^6$	$7 \cdot 10^6$

13 Timing results for real Givens rotations

Six routines to compute real Givens rotations were tested in a way entirely analogous to the manner described in section 10. The test arguments and timing results are shown in the table and figures below.

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

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

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 anyway, so those routines do 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

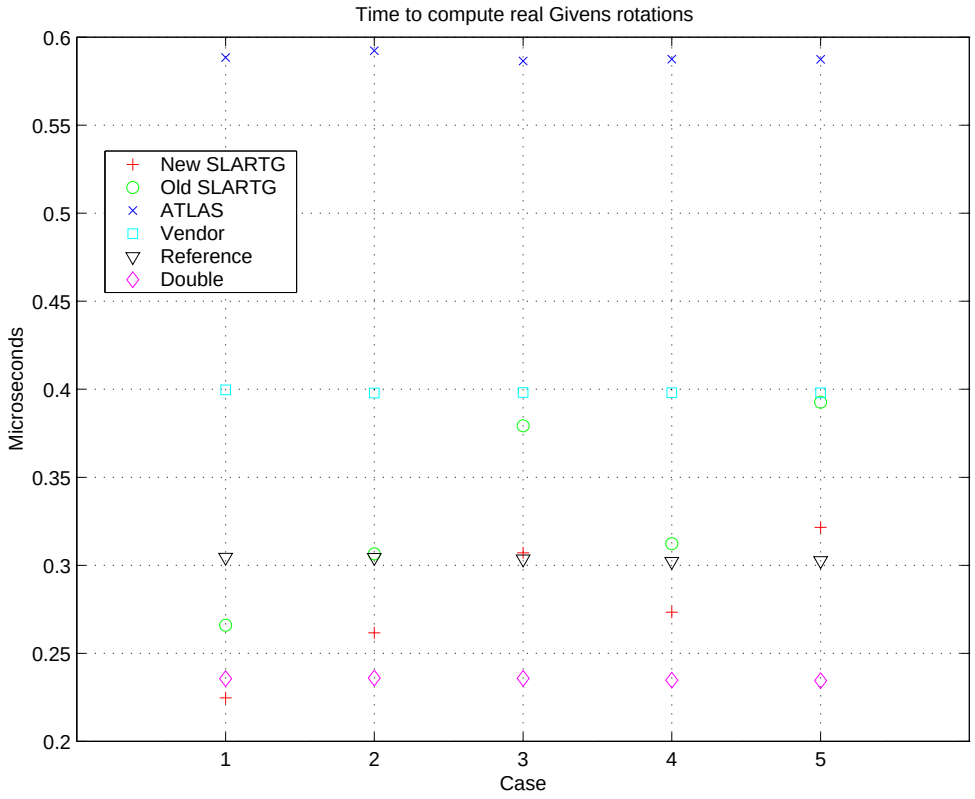

Figure 7: Time to compute real Givens rotations.

We see from figures 7 and 8 that in the most common case (Case 1, where no scaling is needed), the new SLARTG is about 18% faster than the old SLARTG, and 1.35 to 2.62 times faster than any version of SROTG.

To get an absolute speed limit, we also ran a version of the algorithm that only works in Case 1; i.e. 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 .161 microseconds, about 72% 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.

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 nearly as fast as the new SLARTG in the most common case, when no scaling is needed, and faster when scaling is needed.

14 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.

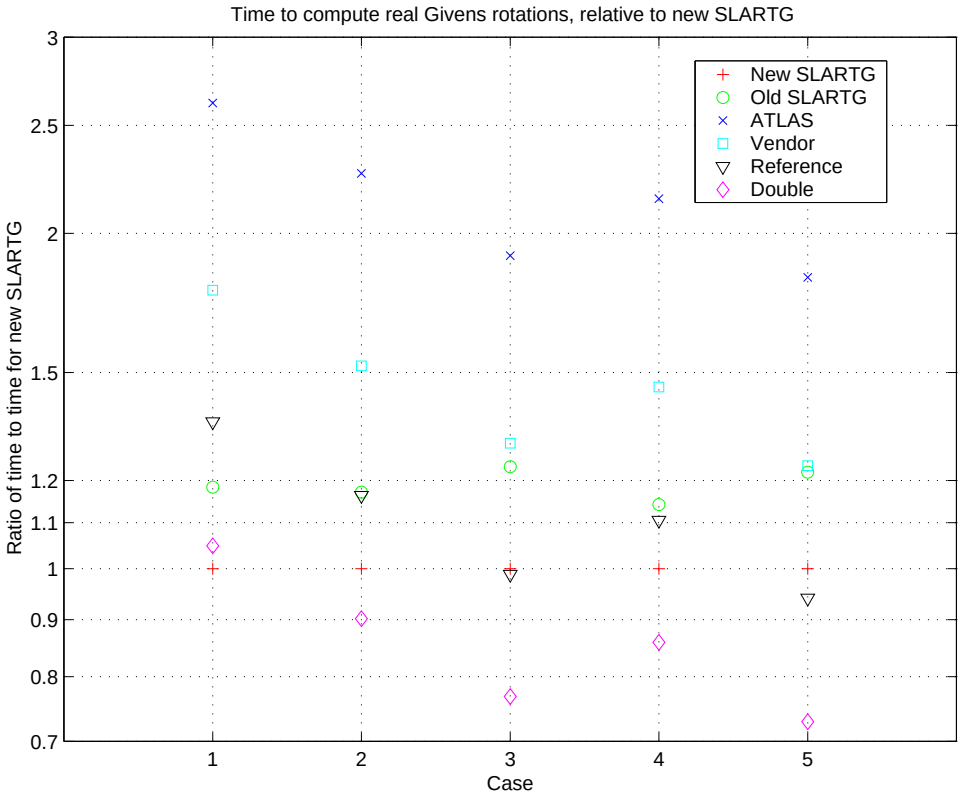

Figure 8: Relative Time to compute real Givens rotations.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide (third edition).* SIAM, Philadelphia, 1999.

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

[3] S. Blackford, G. Corliss, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, C. Hu, W. Kahan, L. Kaufman, B. Kearfott, F. Krogh, X. Li, Z. Maany, A. Petitet, R. Pozo, K. Remington, W. Walster, C. Whaley, and J. Wolff v. Gudenberg. Document for the Basic Linear Algebra Subprograms (BLAS) Standard: BLAS Technical Forum. www.netlib.org/cgi-bin/checkout/blast/blast.pl, 1999.

[4] J. Demmel and X. Li. Faster numerical algorithms via exception handling. *IEEE Trans. Comp.*, 43(8):983–992, 1994. LAPACK Working Note 59.

[5] G. Golub and C. Van Loan. *Matrix Computations.* Johns Hopkins University Press, Baltimore, MD, 3rd edition, 1996.

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

[7] D. Priest. private communication, 2000.

A SLARTG

```
      SUBROUTINE SLARTG( F, G, CS, SN, R )
*
*  -- LAPACK auxiliary routine (version 3.0) --
*     Univ. of Tennessee, Univ. of California Berkeley, NAG Ltd.,
*     Courant Institute, Argonne National Lab, and Rice University
*     Jan 17, 2001
*
*     .. Scalar Arguments ..
      REAL               CS, F, G, R, SN
*     ..
*
*  Purpose
*  =======
*
*  SLARTG generate a plane rotation so that
*
*     [  CS  SN  ]  .  [ F ]  =  [ R ]   where CS**2 + SN**2 = 1.
*     [ -SN  CS  ]     [ G ]     [ 0 ]
*
*  This is a slower, more accurate version of the BLAS1 routine SROTG,
*  with the following other differences:
*     F and G are unchanged on return.
*     If F=0 and G=0, then CS=1, SN=0, and R=0.
*     If F .ne. 0 and G=0, then CS=1, SN=0, and R=F.
*     If F=0 and G .ne. 0, then CS=0, SN=sign(G), and R=abs(G).
*     If F .ne. 0 and (G .ne. 0), then
*             CS = abs(F)/sqrt(F**2 + G**2)
*             SN = sign(F)*G/sqrt(F**2 + G**2)
*             R  = sign(F)*sqrt(F**2 + G**2)
*
*  If a NaN occurs in the input, then R and possibly also
*  CS and SN will be a NaN.
*
*  If an infinity occurs in the input, then R and possibly also
*  CS and SN will be infinite or NaNs.
*
*  The complex routine CLARTG returns the same
*  CS and SN on complex inputs (F,0) and (G,0).
*
*  Arguments
*  =========
*
*  F       (input) REAL
*          The first component of vector to be rotated.
*
*  G       (input) REAL
*          The second component of vector to be rotated.
*
*  CS      (output) REAL
*          The cosine of the rotation.
*
*  SN      (output) REAL
```

```
*          The sine of the rotation.
*
*  R       (output) REAL
*          The nonzero component of the rotated vector.
*
*  =====================================================================
*
*     .. Parameters ..
      REAL               ZERO
      PARAMETER          ( ZERO = 0.0E0 )
      REAL               ONE
      PARAMETER          ( ONE = 1.0E0 )
      REAL               TWO
      PARAMETER          ( TWO = 2.0E0 )
*     ..
*     .. Local Scalars ..
      LOGICAL            FIRST
      INTEGER            COUNT, I, MAXCNT
      REAL               EPS, F1, G1, SAFMIN, SAFMN2, SAFMX2, SCALE
      REAL               SCL, ESFMN2
*     ..
*     .. External Functions ..
      REAL               SLAMCH
      EXTERNAL           SLAMCH
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          ABS, INT, LOG, MAX, SQRT, SIGN
*     ..
*     .. Save statement ..
      SAVE               FIRST, EPS, SAFMX2, SAFMIN, SAFMN2, SAFMN
      SAVE               SAFMX
*     ..
*     .. Data statements ..
      DATA               FIRST / .TRUE. /
*     ..
*     .. Executable Statements ..
*
      IF( FIRST ) THEN
*
*        On first call to SLARTG, compute
*        SAFMN2 = sqrt(SAFMIN/EPS) rounded down to the nearest power
*                 of the floating point radix
*        This means that scaling by multiplication by SAFMN2 and its
*        reciprocal SAFMX2 cause no roundoff error
*
         FIRST = .FALSE.
         SAFMIN = SLAMCH( 'S' )
         EPS = SLAMCH( 'E' )
         ESFMN2 = INT( LOG( SAFMIN / EPS ) / LOG( SLAMCH( 'B' ) ) / TWO)
         SAFMN2 = SLAMCH( 'B' )**ESFMN2
         SAFMN = SAFMN2**2
         SAFMX2 = ONE / SAFMN2
         SAFMX = SAFMX2**2
         MAXCNT = INT( -MAX( SLAMCH('L'), SLAMCH('N')+ONE-SLAMCH('M') )
```

```
     +            /ESFMN2 - ONE - ONE )
      END IF
      IF( G.EQ.ZERO ) THEN
*
*        Includes the case F=G=0
*
         CS = ONE
         SN = ZERO
         R = F
      ELSE IF( F.EQ.ZERO ) THEN
*
*        G must be nonzero
*
         CS = ZERO
         SN = SIGN( ONE, G )
         R  = ABS(G)
      ELSE
*
*        Both F and G must be nonzero
*
         F1 = F
         G1 = G
         SCALE = MAX( ABS( F1 ), ABS( G1 ) )
         COUNT = 0
         IF( SCALE.GE.SAFMX2 ) THEN
*
*           Handle case where F1**2 + G1**2 might overflow
*
            SCL = SAFMX2
*
            COUNT = COUNT + 1
            F1 = F1*SAFMN2
            G1 = G1*SAFMN2
            SCALE = SCALE*SAFMN2
            IF( SCALE.LT.SAFMX2 ) GOTO 100
*
            COUNT = COUNT + 1
            F1 = F1*SAFMN2
            G1 = G1*SAFMN2
            SCALE = SCALE*SAFMN2
            IF( SCALE.LT.SAFMX2 ) GOTO 100
*
10          CONTINUE
            COUNT = COUNT + 1
            F1 = F1*SAFMN2
            G1 = G1*SAFMN2
            SCALE = SCALE*SAFMN2
            IF( SCALE.GT.SAFMX2 .AND. COUNT.LE.MAXCNT ) GOTO 10
         ELSE IF( SCALE.LE.SAFMN2 ) THEN
*
*           Handle case where F1**2 + G1**2 might underflow
*
            SCL = SAFMN2
*
```

```
            COUNT = COUNT + 1
            F1 = F1*SAFMX2
            G1 = G1*SAFMX2
            SCALE = SCALE*SAFMX2
            IF( SCALE.GT.SAFMN2 ) GOTO 100
*
            COUNT = COUNT + 1
            F1 = F1*SAFMX2
            G1 = G1*SAFMX2
            SCALE = SCALE*SAFMX2
            IF( SCALE.GT.SAFMN2 ) GOTO 100
*
20          CONTINUE
            COUNT = COUNT + 1
            F1 = F1*SAFMX2
            G1 = G1*SAFMX2
            SCALE = SCALE*SAFMX2
            IF( SCALE.LT.SAFMN2 .AND. COUNT.LE.MAXCNT ) GOTO 20
         ENDIF
100      CONTINUE
         R = SQRT( F1**2+G1**2 )
         RR = ONE/R
         CS = ABS(F1) * RR
         SN = G1 * RR
         IF (F .LT. ZERO) THEN
            R = -R
            SN = -SN
         ENDIF
         DO 40 I = 1, COUNT
            R = R*SCL
40       CONTINUE
      ENDIF
      RETURN
*
*     End of SLARTG
*
      END
```

B CLARTG

```
      SUBROUTINE CLARTG( F, G, CS, SN, R )
*
*  -- LAPACK auxiliary routine (version 3.0) --
*     Univ. of Tennessee, Univ. of California Berkeley, NAG Ltd.,
*     Courant Institute, Argonne National Lab, and Rice University
*     Jan 10, 2001
*
*     .. Scalar Arguments ..
      REAL               CS
      COMPLEX            F, G, R, SN
*     ..
*
*  Purpose
*  =======
*
*  CLARTG generates a plane rotation so that
*
*     [  CS  SN  ]     [ F ]     [ R ]
*     [  __      ]  .  [   ]  =  [   ]   where CS**2 + |SN|**2 = 1.
*     [ -SN  CS  ]     [ G ]     [ 0 ]
*
*  This is a faster version of the BLAS1 routine CROTG, except for
*  the following differences:
*     F and G are unchanged on return.
*     If F=0 and G=0, then CS=1, SN=0, and R=0.
*     If F .ne. 0 and G=0, then CS=1, SN=0, and R=F.
*     If F=0 and G .ne. 0, then CS=0, SN=conj(G)/abs(G), and R=abs(G).
*     If F .ne. 0 and G .ne. 0, then
*             CS = abs(F)/sqrt(F**2 + G**2)
*             SN = (F/abs(F))*conj(G)/sqrt(F**2 + G**2)
*             R  = (F/abs(F))*sqrt(F**2 + G**2)
*
*  If a NaN occurs in the input, then either real(R) and/or imag(R),
*  and possibly also CS, real(SN) and imag(SN), will be NaNs.
*
*  If an infinity occurs in the input, then either real(R) and/or imag(R),
*  and possibly also CS, real(SN) and imag(SN), will be infinite or NaNs.
*
*  The real routine SLARTG returns the same CS and SN (modulo roundoff)
*  if the inputs F and G are real.
*
*  Arguments
*  =========
*
*  F       (input) COMPLEX
*          The first component of vector to be rotated.
*
*  G       (input) COMPLEX
*          The second component of vector to be rotated.
*
*  CS      (output) REAL
*          The cosine of the rotation.
```

```
*
*  SN      (output) COMPLEX
*          The sine of the rotation.
*
*  R       (output) COMPLEX
*          The nonzero component of the rotated vector.
*
*  =====================================================================
*
*     .. Parameters ..
      REAL               FOUR, ONE, ZERO
      PARAMETER          ( FOUR = 4.0E+0, ONE = 1.0E+0, ZERO = 0.0E+0 )
      COMPLEX            CZERO
      PARAMETER          ( CZERO = ( 0.0E+0, 0.0E+0 ) )
*     ..
*     .. Local Scalars ..
      LOGICAL            FIRST, AGAIN
      INTEGER            COUNT, I, MAXCNT
      REAL               D1, EPS, F2, G2, SAFMIN,
     $                   SAFMN2, SAFMX2, SAFMN4, SAFMX4, SAFMN, SAFMX,
     $                   SCALEF, SCALEG, SCALEFG, FG2, SQREPS, ESFMN4
      COMPLEX            FF, FS, GS
*     ..
*     .. External Functions ..
      REAL               SLAMCH, SLAPY2
      EXTERNAL           SLAMCH, SLAPY2
*     ..
*     .. Intrinsic Functions ..
      INTRINSIC          ABS, AIMAG, CMPLX, CONJG, INT, LOG, MAX, REAL,
     $                   SQRT
*     ..
*     .. Statement Functions ..
      REAL               ABS1, ABSSQ
*     ..
*     .. Save statement ..
      SAVE               FIRST, SAFMIN, EPS, SQREPS
      SAVE               SAFMX2, SAFMX4, SAFMN2, SAFMN4, SAFMN, SAFMX
*     ..
*     .. Data statements ..
      DATA               FIRST / .TRUE. /
*     ..
*     .. Statement Function definitions ..
      ABS1( FF ) = MAX( ABS( REAL( FF ) ), ABS( AIMAG( FF ) ) )
      ABSSQ( FF ) = REAL( FF )**2 + AIMAG( FF )**2
*     ..
*     .. Executable Statements ..
*
      IF( FIRST ) THEN
*
*        On first call to SLARTG, compute
*
*        SAFMN4 = (SAFMIN/EPS)**.25 rounded down to the nearest power
*                 of the floating point radix
*        SAFMN2 = (SAFMIN/EPS)**.5  rounded down to the nearest power
```

```
*                of the floating point radix
*
*        This means that scaling by SAFMN{2,4} and their
*        reciprocals SAFMX{2,4} causes no roundoff error
*
         FIRST = .FALSE.
         SAFMIN = SLAMCH( 'S' )
         EPS = SLAMCH( 'E' )
         SQREPS = SQRT( EPS )
         ESFMN4 = INT( LOG( SAFMIN / EPS ) /
     +                 LOG( SLAMCH( 'B' ) ) / FOUR )
         SAFMN4 = SLAMCH( 'B' )**ESFMN4
         SAFMN2 = SAFMN4**2
         SAFMN  = SAFMN2**2
         SAFMX4 = ONE / SAFMN4
         SAFMX2 = SAFMX4**2
         SAFMX  = SAFMX2**2
*
*        MAXCNT is the maximum number of times a nonzero number
*        can be scaled up/down by SAFMN4 before reaching 1
*
         MAXCNT = INT( -MAX( SLAMCH('L'), SLAMCH('N')+ONE-SLAMCH('M') )
     +            /ESFMN4 - ONE - ONE )
      ENDIF
*
*     If F (or G) contains a NaN, SCALEF (or SCALEG) might not
      SCALEF = ABS1( F )
      SCALEG = ABS1( G )
      IF( SCALEG.EQ.ZERO ) THEN
*
*        Includes the case F=G=0
*        Includes the case F is infinite or NaN:
*          If F is a NaN then R will be a NaN
*          If F is infinite then R will be infinite
*        May include cases where G is a NaN:
*          If G is a NaN then R will be a NaN
*
         CS = ONE
         SN = CZERO
*        Add G to ensure that if G contains a NaN, so does R
         R = F + G
      ELSEIF( SCALEF.EQ.ZERO ) THEN
*
*        G must be nonzero.
*        Includes the cases where G is infinite or NaN:
*          If G is a NaN then SN and R will be NaNs.
*          If G is infinite then SN will be a NAN and R will be infinite
*        May Include cases where F is a NaN:
*          If F is a NaN then R will be a NaN
*
         CS = ZERO
         GS = G
         COUNT = 0
*        No scaling if SCALEG is a NaN
```

```
         IF ( SCALEG .GT. SAFMX2 ) THEN
1           CONTINUE
            COUNT = COUNT + 1
            GS = GS * SAFMN
            SCALEG = SCALEG * SAFMN
*           Keep scaling unless SCALEG is infinite
            IF ( SCALEG .GT. SAFMX2 .AND. COUNT .LE. MAXCNT ) GOTO 1
            SCALE = SAFMX
         ELSEIF( SCALEG .LT. SAFMN2 ) THEN
2           CONTINUE
            COUNT = COUNT + 1
            GS = GS * SAFMX
            SCALEG = SCALEG * SAFMX
*           Keep scaling unless SCALEG=0 because G contains a NaN
            IF ( SCALEG .LT. SAFMN2 .AND. COUNT .LE. MAXCNT ) GOTO 2
            SCALE = SAFMN
         ENDIF
         D1 = SQRT( REAL(GS)**2 + AIMAG(GS)**2 )
         R = D1
         D1 = ONE/D1
         SN = CMPLX( REAL(GS)*D1, -AIMAG(GS)*D1 )
         DO 3 I = 1, COUNT
            R = CMPLX( REAL(R)*SCALE, ZERO )
3        CONTINUE
*        Make sure that R contains a NaN if F does
         R = R + F
      ELSE
*
*        Both F and G must be nonzero
*
         IF( SCALEF.LE.SAFMX4 .AND. SCALEF.GE.SAFMN4 .AND.
     $       SCALEG.LE.SAFMX4 ) THEN
*
*           Case 1: Neither F nor G too big or too small, minimal work
*                   Neither F nor G can be infinite
*                   If either F or G a NaN, then
*                      CS, SR and R will be NaNs
*
            F2 = ABSSQ(F)
            G2 = ABSSQ(G)
            FG2 = F2+G2
            D1 = ONE/SQRT( F2*FG2 )
            CS = F2*D1
            FG2 = FG2 * D1
            R  = CMPLX( REAL(F)*FG2, AIMAG(F)*FG2 )
            SN = CMPLX( REAL(F)*D1 , AIMAG(F)*D1 )
            SN = CONJG(G) * SN
         ELSEIF( SCALEG .LT. SQREPS*SCALEF ) THEN
*
*           Case 2: ABS(F)**2 + ABS(G)**2 rounds to ABS(F)**2
*                   F may be infinite but not G
*                   Either F and/or G may be NaNs
*                   CS = 1 always, even if F and/or G is a NaN
*                   SN is a NaN if F or G is a NaN or infinite.
```

```
*                  R is a NaN (or infinite) if F is a NaN (or infinite).
*
            CS = ONE
            FS = F
            GS = G
            COUNTF = 0
            COUNTG = 0
            IF( SCALEF .GT. SAFMX2 ) THEN
10             CONTINUE
               COUNTF = COUNTF + 1
               FS = FS * SAFMN
               SCALEF = SCALEF * SAFMN
*              Keep scaling unless SCALEF is infinite
               IF ( SCALEF .GT. SAFMX2 .AND. COUNTF .LE. MAXCNT) GOTO 10
            ELSEIF( SCALEF .LT. SAFMN2 ) THEN
20             CONTINUE
               COUNTF = COUNTF - 1
               FS = FS * SAFMX
               SCALEF = SCALEF * SAFMX
*              Keep scaling unless SCALEF=0 because F contains a NaN
               IF ( SCALEF .LT. SAFMN2 .AND. COUNTF.GE.-MAXCNT ) GOTO 20
            ENDIF
            IF( SCALEG .GT. SAFMX2 ) THEN
30             CONTINUE
               COUNTG = COUNTG - 1
               GS = GS * SAFMN
               SCALEG = SCALEG * SAFMN
*              SCALEG finite so scaling must terminate
               IF ( SCALEG .GT. SAFMX2 ) GOTO 30
            ELSEIF( SCALEG .LT. SAFMN2 ) THEN
40             CONTINUE
               COUNTG = COUNTG + 1
               GS = GS * SAFMX
               SCALEG = SCALEG * SAFMX
*              Keep scaling unless SCALEG=0 because G contains a NaN
               IF ( SCALEG .LT. SAFMN2 .AND. COUNTG .LE.MAXCNT ) GOTO 40
            ENDIF
            D1 = ONE/(REAL(FS)**2 + AIMAG(FS)**2)
            SN = FS * CONJG(GS)
*           SN will be a NaN if F is infinite or a NaN
            SN = CMPLX( REAL(SN)*D1 , AIMAG(SN)*D1 )
            COUNT = COUNTF + COUNTG
            IF( COUNT .GT. 0 ) THEN
               DO 50 I = 1, COUNT
                  SN = CMPLX( REAL(SN)*SAFMN , AIMAG(SN)*SAFMN )
50             CONTINUE
            ELSEIF( COUNT .LT. 0 ) THEN
               DO 60 I = 1, -COUNT
                  SN = CMPLX( REAL(SN)*SAFMX , AIMAG(SN)*SAFMX )
60             CONTINUE
            ENDIF
*           Make sure R contains a NaN if G does
            R = F + SN*G
         ELSEIF( SCALEF .LT. SQREPS*SCALEG ) THEN
```

```
*
*           Case 3: ABS(F)**2 + ABS(G)**2 rounds to ABS(G)**2
*                   G may be infinite but not F
*                   Either F or G may be a NaN, in which case
*                   CS, SN and R are NaNs
*                   SN and R are NaNs if G is infinite
*
            FS = F
            GS = G
            COUNTF = 0
            COUNTG = 0
            IF( SCALEF .GT. SAFMX4 ) THEN
70             CONTINUE
               COUNTF = COUNTF + 1
               FS = FS * SAFMN2
               SCALEF = SCALEF * SAFMN2
*              SCALEF finite so scaling must terminate
               IF ( SCALEF .GT. SAFMX4 ) GOTO 70
            ELSEIF( SCALEF .LT. SAFMN4 ) THEN
80             CONTINUE
               COUNTF = COUNTF - 1
               FS = FS * SAFMX2
               SCALEF = SCALEF * SAFMX2
*              Keep scaling unless SCALEF=0 because F contains a NaN
               IF ( SCALEF .LT. SAFMN4 .AND. COUNTF .GE.-MAXCNT) GOTO 80
            ENDIF
            IF( SCALEG .GT. SAFMX4 ) THEN
90             CONTINUE
               COUNTG = COUNTG + 1
               GS = GS * SAFMN2
               SCALEG = SCALEG * SAFMN2
*              Keep scaling unless SCALEG is infinite
               IF ( SCALEG .GT. SAFMX4 .AND. COUNTG .LE. MAXCNT) GOTO 90
            ELSEIF( SCALEG .LT. SAFMN4 ) THEN
100            CONTINUE
               COUNTG = COUNTG - 1
               GS = GS * SAFMX2
               SCALEG = SCALEG * SAFMX2
*              SCALEG cannot be zero so scaling must terminate
               IF ( SCALEG .LT. SAFMN4 ) GOTO 100
            ENDIF
            F2 = REAL(FS)**2 + AIMAG(FS)**2
            G2 = REAL(GS)**2 + AIMAG(GS)**2
            D1 = ONE/SQRT( F2*G2 )
            CS = F2*D1
            SN = FS * CONJG(GS)
*           SN will be a NaN if G is infinite
            SN = CMPLX( REAL(SN)*D1 , AIMAG(SN)*D1 )
            D1 = G2*D1
*           R will be a NaN if G is infinite
            R  = CMPLX( REAL(FS)*D1 , AIMAG(FS)*D1 )
            COUNT = COUNTF - COUNTG
            IF( COUNT .GT. 0 ) THEN
               DO 110 I = 1, COUNT
```

```
               CS = CS*SAFMX2
110            CONTINUE
            ELSEIF( COUNT .LT. 0 ) THEN
               DO 120 I = 1, -COUNT
                  CS = CS*SAFMN2
120            CONTINUE
            ENDIF
            IF( COUNTG .GT. 0 ) THEN
               DO 130 I = 1, COUNTG
                  R = CMPLX( REAL(R)*SAFMX2, AIMAG(R)*SAFMX2 )
130            CONTINUE
            ELSEIF( COUNTG .LT. 0 ) THEN
               DO 140 I = 1, -COUNTG
                  R = CMPLX( REAL(R)*SAFMN2, AIMAG(R)*SAFMN2 )
140            CONTINUE
            ENDIF
         ELSE
*
*           Case 4: Scale F and G up or down and use formula from Case 1
*                   Both F and G may be simultaneously infinite,
*                     in which case CS, SN and R are all NaNs
*                   Either F or G may be a NaN, in which case
*                   CS, SN and R are all NaNs
*
            FS = F
            GS = G
            COUNT = 0
            AGAIN = .FALSE.
            SCALEFG = MAX( SCALEF, SCALEG )
            IF( SCALEFG .GT. ONE) THEN
*
               IF (SCALEFG .LE. SAFMX2) THEN
                  SCALE2 = SAFMX4
                  AGAIN = .TRUE.
                  FS = FS * SAFMN4
                  GS = GS * SAFMN4
                  GOTO 153
               ENDIF
               SCALE = SAFMX2
               FS = FS * SAFMN2
               GS = GS * SAFMN2
               SCALEFG = SCALEFG * SAFMN2
               COUNT = COUNT + 1
*
*              Keep scaling unless SCALEFG is infinite
151            CONTINUE
               IF( SCALEFG .LE. SAFMX2 .OR. COUNT .GT. MAXCNT) GOTO 152
                  FS = FS * SAFMN2
                  GS = GS * SAFMN2
                  SCALEFG = SCALEFG * SAFMN2
                  COUNT = COUNT + 1
                  GOTO 151
152            CONTINUE
               IF( SCALEFG .GT. SAFMX4) THEN
```

```
                  SCALE2 = SAFMX4
                  AGAIN = .TRUE.
                  FS = FS * SAFMN4
                  GS = GS * SAFMN4
               ENDIF
            ELSE
*              SCALEFG might be a NaN
*
               IF (SCALEF .GE. SAFMN2) THEN
                  SCALE2 = SAFMN4
                  AGAIN = .TRUE.
                  FS = FS * SAFMX4
                  GS = GS * SAFMX4
                  GOTO 153
               ENDIF
               SCALE = SAFMN2
               FS = FS * SAFMX2
               GS = GS * SAFMX2
               SCALEF = SCALEF * SAFMX2
               COUNT = COUNT + 1
*
160            CONTINUE
               IF( SCALEF .GE. SAFMN2 .OR. COUNT .GT. MAXCNT ) GOTO 162
                  FS = FS * SAFMX2
                  GS = GS * SAFMX2
                  SCALEF = SCALEF * SAFMX2
                  COUNT = COUNT + 1
162            CONTINUE
               IF( SCALEF .LT. SAFMN4 ) THEN
                  SCALE2 = SAFMN4
                  AGAIN = .TRUE.
                  FS = FS * SAFMX4
                  GS = GS * SAFMX4
               ENDIF
            ENDIF
*
153         CONTINUE
            F2 = ABSSQ(FS)
            G2 = ABSSQ(GS)
            FG2 = F2+G2
            D1 = ONE/SQRT( F2*FG2 )
*           CS will be a NaN if both F and G infinite
            CS = F2*D1
            FG2 = FG2 * D1
            R  = CMPLX( REAL(FS)*FG2, AIMAG(FS)*FG2 )
            SN = CMPLX( REAL(FS)*D1 , AIMAG(FS)*D1 )
*           SN will be a NaN if both F and G infinite
            SN = CONJG(GS) * SN
            DO 170 I = 1, COUNT
               R = CMPLX( REAL(R) * SCALE, AIMAG(R) * SCALE )
170         CONTINUE
            IF ( AGAIN )
     $         R = CMPLX( REAL(R) * SCALE2, AIMAG(R) * SCALE2 )
         ENDIF
```

```
      ENDIF
      RETURN
*
*     End of CLARTG
*
      END
```