Algorithm 978: Safe Scaling in the Level 1 BLAS

EDWARD ANDERSON, Lockheed Martin

The square root of a sum of squares is well known to be prone to overflow and underflow. Ad hoc scaling of intermediate results, as has been done in numerical software such as the BLAS and LAPACK, mostly avoids the problem, but it can still occur at extreme values in the range of representable numbers. More careful scaling, as has been implemented in recent versions of the standard algorithms, may come at the expense of performance or clarity. This work reimplements the vector 2-norm and the generation of Givens rotations from the Level 1 BLAS to improve their performance and design. In addition, support for negative increments is extended to the Level 1 BLAS operations on a single vector, and a comprehensive test suite for all the Level 1 BLAS is included.

CCS Concepts: • **Mathematics of computing** → *Mathematical software performance*; • **Software and its engineering** → *Software libraries and repositories*

Additional Key Words and Phrases: Vector operation, sum of squares, Givens rotation

ACM Reference Format:
Edward Anderson. 2017. Algorithm 978: Safe scaling in the level 1 BLAS. ACM Trans. Math. Softw. 44, 1, Article 12 (July 2017), 28 pages.
DOI: http://dx.doi.org/10.1145/3061665

1. INTRODUCTION

The computation of the 2-norm of a vector x of length n,

$$||x||_2 = \left(\sum_{i=1}^{n} x_i^2\right)^{1/2},$$

is a fundamental operation in numerical linear algebra software. The 2-norm or just a sum of squares can also occur in other contexts where the software designer may not think to call a library routine, such as complex division,

$$\frac{u}{v} = \frac{a+bi}{c+di} = \frac{(ac+bd)+(bc-ad)\,i}{c^2+d^2},$$

and the calculation of the sine of the angle between the x-axis and the line from (0,0) to (a, b), in the $x - y$ plane:

$$s = \frac{b}{\sqrt{a^2+b^2}}.$$

The difficulty in computing these quantities is that the sum of squares of the elements of a vector x may overflow if any $|x_i|$ is greater than the square root of the largest

Author's address: E. Anderson, U.S. EPA, N127-01, 109 T.W. Alexander Drive, Research Triangle Park, NC 27711; email: Anderson.Edward@epa.gov.

DOI: http://dx.doi.org/10.1145/3061665

representable number, and it may underflow to zero if every $|x_i|$ is less than the square root of the smallest representable number unless a suitable scaling is used. Software to implement these operations, in the words of James Blue [1978], "should be accurate and efficient, and should avoid all overflows and underflows," but these objectives have not always been achieved.

1.1. Issues with the Reference BLAS

A function to compute the 2-norm of a vector x of length n was included as part of the original Basic Linear Algebra Subprograms (BLAS) [Lawson et al. 1979] using an algorithm based on work by Blue [1978]. Blue's algorithm divides the real number line into small, medium, and large number ranges and uses three accumulators when computing the sum of squares, one for the small numbers, which have to be scaled up before being squared; one for the medium numbers, which do not require scaling; and one for the large numbers, which have to be scaled down before being squared. The implementation in the BLAS was efficient but difficult to understand, leading many library writers to write their own, but these alternative versions were not always as careful about scaling. In the 1990s, the reference version on netlib [Dongarra and Grosse 1987] was replaced with a single-pass algorithm from LAPACK [Anderson et al. 1999] that makes one pass through the vector and continually rescales the sum by the largest element in absolute value. This method avoids overflow by generally keeping the sum of squares between 1.0 and n, but it scales every element, whether it needs scaling or not, and the rescaling inhibits vectorization, making the algorithm very slow [Anderson and Fahey 1996]. Also, if two or more of the elements of x are $\pm\infty$, the norm is computed as NaN, when it should be ∞.

Scaling is also required in the BLAS procedure to generate plane rotations, commonly called Givens rotations. A square root of a sum of squares occurs when computing the sine and cosine of the angle of rotation. In the subroutine with real valued inputs from the BLAS and its complex analog from LINPACK [Dongarra et al. 1979], the components a and b of the sum of squares $a^2 + b^2$ were divided by $|a| + |b|$ without checking if this quantity was invertible or even computable without overflow. The complex algorithm also did not return the same result as the real algorithm if provided with real valued inputs. Bindel et al. [2002] analyze the scaling requirements for constructing Givens rotations in great detail, but their implementation [Demmel 2002] fails to avoid all the scaling pitfalls and is unable to pass the newly constructed tests.

The remaining vector (Level 1) BLAS have a few minor issues. The reference versions employ manual loop unrolling to a depth of up to 7 based on previous studies showing a performance benefit [Dongarra 1980], but manual unrolling is unnecessary now, because it is a standard option in optimizing compilers. The reference Level 1 BLAS are also inconsistent in their handling of vector increments, allowing the increment to be positive, negative, or zero in some subroutines but only allowing positive increments in others. Also, the Level 1 BLAS test program is woefully inadequate, including only a few small test cases with precomputed results, and no vector longer than five elements.

1.2. Features of the New Model Implementation

In this work, we reimplement Blue's algorithm for the 2-norm of a vector following the original specification and show that it is faster than the current reference version. For the generation of Givens rotations, we present a simpler algorithm than the one in LAPACK and show that its scaling is sufficient under very mild assumptions about the floating-point model. To address deficiencies in the other Level 1 BLAS, we introduce a new model implementation in a more contemporary programming style, remove the manual unrolling, extend the interface to admit general vector increments for all subroutines, and construct a more comprehensive and scalable test suite.

A few possible extensions are omitted from this study. The procedures for generating and applying modified Givens rotations are not included, although they were part of the original BLAS. A modified Givens rotation refactors the 2×2 Givens rotation matrix as the product of a diagonal matrix and a 2×2 orthogonal transformation matrix with some of its entries either 1 or −1, saving work when the rotation is applied if the diagonal scaling does not need to be applied to the row vectors. The subroutines to generate a modified Givens rotation were modernized using features of Fortran 90 by Hopkins [1997], while subroutines that both generate and apply a modified Givens transformation, replacing the legacy BLAS equivalents, were proposed by Hanson and Hopkins [2004], so it is not clear that reimplementing the legacy interface to these subroutines would have any value. Also, the matrix-vector (Level 2) BLAS [Dongarra et al. 1988] and the matrix-matrix (Level 3) BLAS [Dongarra et al. 1990] are not included, because their reference implementations do not have the same issues identified in the vector (Level 1) BLAS; there is no manual unrolling, all the vector increments are fully general, and the existing test suites already allow test problem sizes to be controlled by an input file.

Section 2 defines the scaling constants used in the new model implementation in terms of the floating-point model. Section 3 describes Blue's algorithm for the vector 2-norm and the modifications made to adapt it to contemporary floating-point models. Section 4 revives a simpler algorithm for computing Givens rotations, introduces a new complex equivalent, and shows that these implementations avoid overflow and underflow under mild assumptions about the floating-point model. Section 5 describes a new model implementation of the Level 1 BLAS and a few LAPACK extensions and a new comprehensive test program for them. Section 6 contains some concluding remarks and suggestions for further development.

2. SCALING TO AVOID UNDERFLOW AND OVERFLOW

Underflow and overflow are defined by the floating-point model. The floating-point number system used in Fortran and described by Blue [1978] is a subset of the real numbers consisting of zero and all elements of the form

$$f = \pm(m_1\beta^{-1} + \cdots m_t\beta^{-t}) \times \beta^e,$$

where

$$1 \le m_1 < \beta; 0 \le m_i < \beta, i = 2, \ldots, t;\ e_{min} \le e \le e_{max}.$$

The requirement is that $m_1 \ge 1$ makes the numbers normalized and guarantees that the representation is unique. A floating-point model is defined by four integers: the base or radix β, the number of mantissa digits t, and the minimum and maximum exponents e_{min} and e_{max}, from which one can define the following constants:

Smallest model number:	$r = \beta^{e_{min}-1}$
Largest model number:	$R = \beta^{e_{max}}\left(1 - \beta^{-t}\right)$
Machine precision:	$\varepsilon = \beta^{1-t}$

The smallest model number r is the underflow threshold, while the largest model number R is the overflow threshold. The machine precision is the relative spacing between consecutive model numbers. Sometimes this term is used interchangeably with "machine epsilon," but if the machine epsilon is defined as the maximum relative representation error, then the machine epsilon is half the machine precision in rounded arithmetic [Demmel 1997]. In Fortran, the floating-point model parameters β, t, e_{min}, and e_{max} can be obtained from the intrinsic functions RADIX, DIGITS, MINEXPONENT, and MAXEXPONENT, and the model numbers r, R, and ε can be obtained from the intrinsic functions TINY, HUGE, and EPSILON.

The IEEE floating-point model [IEEE 2008] extends the model above by allowing $m_1 = 0$ if $e = e_{min}$. Numbers with this representation are called denormalized. The inclusion of denormalized numbers allows a more gradual underflow to zero, instead of treating all values less than $\beta^{e_{min}-1}$ as zero. The smallest representable number including the denormalized numbers is $\beta^{e_{min}-t}$.

2.1. Safe and Well-Scaled Numbers

A floating-point number f will be said to be *safe* if $1/f$ can be computed without overflow or underflow. For the purpose of this definition, the value of f will be considered to have underflowed if it is less than the smallest model number, even though it may still be representable as a denormalized number. A floating-point number f will be said to be *well scaled* if f^2 can be computed without overflow or loss of precision due to underflow. Bounds for testing that f is safe or well scaled are discussed in the following section. When discussing only the upper bound, we will say that f is safe from overflow, and when discussing only the lower bound, we will say that f is safe from underflow.

2.2. Boundaries for Scaling Regions

The first constants that we need for scaling are the boundaries SAFMIN and SAFMAX of the range of safe numbers, such that $1/f$ is computable as a model number if SAFMIN $\leq |f| \leq$ SAFMAX. Although in theory the floating-point model parameters could all be chosen independently, in practice the exponent range is usually fairly symmetric about 0, with a slight bias towards positive exponents because overflow is a more serious concern than underflow. Assuming a positive exponent bias, the smallest safe number greater than zero is the smallest model number $\beta^{e_{min}-1}$, and its inverse is $\beta^{1-e_{min}}$. However, if $1 - e_{min} \geq e_{max}$, we would instead set SAFMAX to $\beta^{e_{max}-1}$ and SAFMIN to $\beta^{1-e_{max}}$.

We also require the boundaries of the range of well-scaled numbers for deciding when the sum of squares can be computed without scaling. Extending the definition of a well-scaled number, we will say that a vector x of length n is well scaled if $\sum_{i=1}^{n} x_i^2$ can be computed by an unscaled algorithm without underflow or overflow. More precisely, x is well scaled if $r \leq \sum_{i=1}^{n} x_i^2 \leq R$, a definition that allows some $|x_i| < r$ but not all $|x_i| < r$. A sufficient condition for x to be well scaled is $\sqrt{r} \leq |x_i| \leq \sqrt{R/n}$ for $1 \leq i \leq n$ or $\sqrt{r} \leq |x_i| \leq \sqrt{R \cdot \varepsilon}$ for $n \leq 1/\varepsilon$.

Since the square root may not be exactly computable, Blue [1978] took the nearest power of the radix and defined $t_{sml} = \beta^{\lceil (e_{min}-1)/2 \rceil}$ and $t_{big} = \beta^{\lfloor (e_{max}-t+1)/2 \rfloor}$, where $\lceil u \rceil$ is the smallest integer greater than or equal to u and $\lfloor u \rfloor$ is the largest integer less than or equal to u. Then x is well scaled if $t_{sml} \leq |x_i| \leq t_{big}$ for $1 \leq i \leq n$ and $n \leq 1/\varepsilon$. If any x_i were not well scaled, then Blue employed powers of the radix as scaling constants, multiplying small values by $s_{sml} = \beta^{-\lfloor (e_{min}-1)/2 \rfloor}$ and large values by $s_{big} = \beta^{-\lceil (e_{max}-t+1)/2 \rceil}$. This scaling is sufficient to avoid underflow if r is the smallest representable number, but when denormalized numbers are included, it is necessary to shift some more. Inserting a factor of $1/\sqrt{\varepsilon}$ makes $s_{sml} = \beta^{-\lfloor (e_{min}-t)/2 \rfloor}$, then as desired

$$(x_i \cdot s_{sml})^2 \geq \left(\beta^{e_{min}-t} \cdot \beta^{-(e_{min}-t)/2}\right)^2 \geq \beta^{e_{min}-t}.$$

Note that multiplying values of x near $\sqrt{r}$ by $1/\sqrt{r \cdot \varepsilon}$ makes the scaled value as big as $1/\sqrt{\varepsilon}$, its square as big as $1/\varepsilon$, and the summation of $n \leq 1/\varepsilon$ of them as big as $1/\varepsilon^2$, but this should not cause any difficulty as long as $1/\varepsilon^2 \leq R$, which generally holds if $\beta^{2(t-1)} \leq \beta^{1-e_{min}}$ or $t - 1 \leq (1 - e_{min})/2$. This important model assumption is discussed further in Section 4.4.

Table I. Scaling Constants in Terms of Model Parameters β, t, e_{min}, and e_{max}

Name	Symbol	Formula	Notes
ULP	ε	β^{1-t}	unit in last place
SAFMIN		$\max\left(\beta^{e_{min}-1}, \beta^{1-e_{max}}\right)$	
SAFMAX		$\min\left(\beta^{1-e_{min}}, \beta^{e_{max}-1}\right)$	1/SAFMIN
RTMIN		$(\text{SAFMIN}/\text{ULP})^{1/2}$	
RTMAX		$(\text{SAFMAX} \cdot \text{ULP})^{1/2}$	1/RTMIN
TSML	t_{sml}	$\beta^{\lceil (e_{min}-1)/2 \rceil}$	called b by Blue
TBIG	t_{big}	$\beta^{\lfloor (e_{max}-t+1)/2 \rfloor}$	called B by Blue
USML	u_{sml}	$\beta^{\lfloor (e_{min}-t)/2 \rfloor}$	called s by Blue
UBIG	u_{big}	$\beta^{\lceil (e_{max}-t+1)/2 \rceil}$	called S by Blue
SSML	s_{sml}	$\beta^{-\lfloor (e_{min}-t)/2 \rfloor}$	1/USML
SBIG	s_{big}	$\beta^{-\lceil (e_{max}-t+1)/2 \rceil}$	1/UBIG

Finally, we require the boundaries of the range of numbers that are well scaled and whose square or sum of squares is also safe. These parameters would be needed, for example, in complex division where we have a sum of squares in the denominator. For these boundaries, we divide SAFMIN by ε (also called ULP for "unit in the last place"), multiply SAFMAX by ε, and take their square roots, hence the names RTMIN and RTMAX. The variable names ULP, SAFMIN, SAFMAX, RTMIN, and RTMAX are already in common use in LAPACK.[1] The scaling boundaries and constants are summarized in Table I, and values of these constants for the IEEE 32-bit and 64-bit binary floating-point models are shown in the Appendix.

2.3. Safe Scaling and Blue's Scaling

Consider the special case of computing $s = \sqrt{x^2 + y^2}$ for real inputs x and y. A simple way to avoid overflow or underflow would be to scale x and y by $w = \max(|x|, |y|)$, making $s = w\sqrt{(x/w)^2 + (y/w)^2}$. At the time of this writing, this is exactly what is done in the LAPACK function SLAPY2 in single precision (or DLAPY2 in double precision), the body of which in pseudo code is as shown in Algorithm 1a:

ALGORITHM 1a: LAPACK method for $s = \sqrt{x^2 + y^2}$ with x and y real

```
$w = \max(|x|, |y|)$
$z = \min(|x|, |y|)$
if $z = 0$
    $s = w$
else
    $s = w \cdot \sqrt{1 + (z/w)^2}$
end if
```

There are several problems with this algorithm. First, the Fortran functions *max* and *min* are indeterminate if one of x or y is NaN (not a number). Logically, they should return NaN, but they may not, so if $x = 0$ and y is NaN, it is possible that both w and z could be zero and SLAPY2 could return 0, not NaN. Second, w may not be invertible. For example, if w and z are Inf (a representation of infinity), then z/w will be NaN, making s NaN, not Inf as expected. Finally, there is no attempt to avoid dividing by

[1]We have used the variable name ULP to refer to machine precision instead of EPS, because the machine epsilon is also used in LAPACK. The inquiry function SLAMCH('Precision') returns β^{1-t} while SLAMCH('Epsilon') returns $\frac{1}{2}\beta^{1-t}$ in rounded arithmetic. The symbol ε is used for consistency with Blue.

w when x and y are already well scaled, even though the divide tends to be a costly operation on modern microprocessors. It is also useless to do any scaling if x or y are NaN, but we want to avoid the cost of testing for NaNs, so the main concern is to make sure that the scaling constant is safe if x and y are not NaN.

A safe scaling constant would be

$$w = \min(\text{SAFMAX}, \max(\text{SAFMIN}, |\text{x}|, |\text{y}|)).$$

We call this restriction *safe scaling* because the alternative would be unsafe—we never want to divide by more than SAFMAX or by less than SAFMIN. The algorithm with these improvements is as shown in Algorithm 1b:

ALGORITHM 1b: Safe scaling method for $s = \sqrt{x^2 + y^2}$ with x and y real

```
if $|x| = 0$
    $s = |y|$
else if $|y| = 0$
    $s = |x|$
else

    $w = \min(\text{SAFMAX}, \max(\text{SAFMIN}, |x|, |y|))$
    if $(\text{RTMIN} < |x| < \text{RTMAX})$ and $(\text{RTMIN} < |y| < \text{RTMAX})$
        $s = \sqrt{x^2 + y^2}$
    else
        $s = w \cdot \sqrt{(x/w)^2 + (y/w)^2}$
    end if
end if
```

Here we have not tried to take advantage of the fact that x/w or y/w may be exactly 1 if $w = |x|$ or $w = |y|$. This ensures that any NaNs will propagate to the result.

The final improvement is to use Blue's scaling constants if x or y is not well scaled. Blue did not actually implement this special case, but it would look something like Algorithm 1c:

ALGORITHM 1c: Blue's scaling method for $s = \sqrt{x^2 + y^2}$ with x and y real

```
if $|x| = 0$
    $s = |y|$
else if $|y| = 0$
    $s = |x|$
else
    $w = \max(|x|, |y|)$
    $z = \min(|x|, |y|)$
    if $w > t_{big}$
        $s = (1/s_{big}) \cdot \sqrt{(x \cdot s_{big})^2 + (y \cdot s_{big})^2}$
    else if $w < t_{sml}$
        $s = (1/s_{sml}) \cdot \sqrt{(x \cdot s_{sml})^2 + (y \cdot s_{sml})^2}$
    else if $z < t_{sml}$
        $s = w \cdot \sqrt{1 + (z/w)^2}$
    else
        $s = \sqrt{x^2 + y^2}$
    end if
end if
```

In the actual implementation, the precomputed constants $u_{big} = 1/s_{big}$ and $u_{sml} = 1/s_{sml}$ are used. When either x or y is large, both values are scaled down by a precomputed factor, and when both x and y are small, both are scaled up by a precomputed factor. If only one of x or y is small, the smaller element is scaled by the larger element to avoid a loss of precision when the square of one element is just above the underflow threshold and the square of the other element is just below the underflow threshold and would underflow to zero. The advantage to Blue's scaling is that it does not introduce any roundoff error when multiplying by the precomputed constants because they are powers of the radix.

2.4. Importing Scalar Parameters

The traditional means of getting floating-point constants into a Fortran procedure has been through an inquiry function. LAPACK uses an inquiry function SLAMCH (or DLAMCH in double precision), while its predecessors used functions such as I1MACH/R1MACH/D1MACH [Fox et al. 1978] or MACHAR [Cody 1988]. Even though xLAMCH can be implemented using calls to intrinsic functions provided in Fortran 90, there is still the overhead of the function call itself. To eliminate this overhead, some subroutines in LAPACK compute the constants they need on the first call and store them in a SAVE block for direct use on subsequent calls, a poor programming practice that inhibits thread safety.

A better solution implemented in LAPACK3E [Anderson 2002] is to store constants as parameters in a Fortran module and USE the module. Then there is no overhead of a function call and no reason to duplicate trivial declarations of common constants such as zero and one. For clarity, the constants that are required can be itemized on the USE line, for example,

```
use LA_CONSTANTS, only: wp, zero, one
```

The only disadvantage is that the constants module has to be generated once when the software is compiled. Code to generate the module and versions of the Fortran module source generated on an IEEE-754 system for 32-bit and 64-bit floating-point precision, which most people will be able to use without modification, are provided with the test package and included in the Appendix.

Although the main reason for placing all constants in a module is to allow a more uniform software design, there is also a performance advantage to storing them as parameters with preset values. For example, consider again the LAPACK function SLAPY2 to compute $s = \sqrt{x^2 + y^2}$ as described in the previous section. Three versions of the "safe scaling" version of this function were prepared, one in which the constants required for scaling are obtained from a module, one in which they are obtained from the LAPACK inquiry function or computed, and one in which they are computed on entry using Fortran intrinsic functions for the model parameters. In the module approach, the constants are included via a single USE line:

```
use LA_CONSTANTS32, only: wp, zero, safmin, safmax, rtmin, rtmax
```

while the LAPACK approach using an inquiry function is

```
integer, parameter :: wp = 4
real(wp), parameter :: zero = 0.0_wp
real(wp), parameter :: one = 1.0_wp
real(wp) :: ulp, safmin, safmax, smlnum, rtmin, rtmax
real(wp) :: SLAMCH
```

```
ulp = SLAMCH('Precision')
safmin = SLAMCH('Safe minimum')
safmax = one / safmin
smlnum = safmin / ulp
rtmin = sqrt(smlnum)
rtmax = one / rtmin
```

and the intrinsic function method is

```
integer, parameter :: wp = 4
real(wp), parameter :: zero = 0.0_wp
real(wp), parameter :: one = 1.0_wp
real(wp), parameter :: ulp = epsilon(one)
real(wp), parameter :: safmin = tiny(one)
real(wp), parameter :: safmax = one / safmin
real(wp), parameter :: smlnum = safmin / ulp
real(wp), parameter :: rtmin = sqrt(smlnum)
real(wp), parameter :: rtmax = one / rtmin
```

Each version of SLAPY2 was called repeatedly in a loop with elements from vectors x and y of length $n = 1{,}000{,}000$ and the average time for the n calls was computed. On a single core of an Intel E5–2670 processor, using Intel Fortran version 17.0 with the options `-O3 -fp-model precise`, the following timings were observed:

Method for importing constants	Time for n calls to SLAPY2
Module	4.5 milliseconds
Inquiry function	94.3 milliseconds
Intrinsics	4.5 milliseconds

The performance advantage of the module approach over the inquiry function is clear, and although there is no difference in performance between the module and the intrinsics, there is a huge advantage in software maintainability. In practice, having the same constant computed in many different locations as was done in LAPACK leads to many different values being used for a constant like smlnum. The module approach is used in the new model implementations that accompany this article.

3. SUM OF SQUARES

The sum of squares operation for a vector x with n elements is

$$S = \sum_{i=1}^{n} x_i^2 = x_1^2 + x_2^2 + \cdots + x_n^2,$$

from which the Euclidean or 2-norm can be computed as $||x||_2 = \sqrt{S}$. This computation will overflow if the absolute value of any x_i is greater than the square root of overflow, and it will underflow to zero if the absolute value of every x_i is less than the square root of underflow. Even if some of the x_i's are large enough in magnitude to avoid underflow in $||x||_2$, accuracy may be lost if the contributions of many other x_i's are lost due to underflow. To avoid overflow, values that are large in magnitude must be scaled towards zero, and to avoid underflow, values that are small in magnitude must be scaled away from zero. A further consideration is to ensure that all scaling constants are invertible, so that division can be replaced by multiplication by the reciprocal. But if the calculation does not cause any underflow or overflow, we can and should use an unscaled algorithm.

One approach to scaling is to return a multiplier w and sum of squares q such that $S = w^2 \cdot q$, from which the 2-norm can be computed as $||x||_2 = w \cdot \sqrt{q}$. The LAPACK auxiliary routine xLASSQ computes w and q without forming the 2-norm and also accepts input values for w and q, allowing the sum of squares of the elements of a vector x to be added to an existing sum of squares.

3.1. Blue's Algorithm

To compute $s = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$, Blue used the scaling constants of Section 2.2 to define the ranges of small, medium, and large numbers and employed three accumulators, one for the small numbers that need to be scaled up, one for the medium numbers that do not require scaling, and one for the large numbers that need to be scaled down. The main loop of Blue's algorithm, with the accumulators called a_{sml}, a_{med}, and a_{big}, respectively, is as follows:

```
$a_{sml} = 0; a_{med} = 0; a_{big} = 0$
for $i = 1$ to $n$
    if $|x_i| > t_{big}$
        $a_{big} \leftarrow a_{big} + (x_i \cdot s_{big})^2$
    else if $|x_i| < t_{sml}$
        $a_{sml} \leftarrow a_{sml} + (x_i \cdot s_{sml})^2$
    else
        $a_{med} \leftarrow a_{med} + x_i^2$
    end if
end for
```

Note that any NaN values will fall through the tests and be added to a_{med}, which will then become NaN.

It now remains to combine the accumulators and take the square root. If there is any contribution from a_{big}, then a_{sml} can be ignored, so at most two of the accumulators need to be combined. If a_{big} and a_{med} are nonzero, we apply the scaling factor s_{big} to a_{med} and return

$$s = (1/s_{big}) \cdot \sqrt{a_{big} + (a_{med} \cdot s_{big}) \cdot s_{big}}.$$

If a_{big} is zero but a_{med} and a_{sml} are both nonzero, we cannot add a_{med} into a_{sml} by computing $(a_{med} \cdot s_{sml}) \cdot s_{sml}$ because it would likely cause overflow, and we cannot add a_{sml} into a_{med} by computing $(a_{sml}/s_{sml})/s_{sml}$ without likely causing underflow. Instead we turn this into a SLAPY2 problem with $x = \sqrt{a_{med}}$ and $y = \sqrt{a_{sml}}/s_{sml}$. Clearly, x as a 2-norm of medium numbers cannot be small, so we can use the LAPACK algorithm from Section 2.3 without risk of overflow:

$$\begin{aligned} y_{max} &= \max\left(\sqrt{a_{med}}, \sqrt{a_{sml}}/s_{sml}\right) \\ y_{min} &= \min\left(\sqrt{a_{med}}, \sqrt{a_{sml}}/s_{sml}\right) \\ s &= y_{max} \cdot \sqrt{1 + (y_{min}/y_{max})^2}. \end{aligned}$$

A few modifications have been made to Blue's algorithm in the current implementation. The test to see if a_{med} is nonzero, which will fail if a_{med} is NaN, has been augmented with a check for NaN values to ensure that NaN values propagate to the result. The *isnan* function is implemented as a generic function in the new model implementation using built-in intrinsics if available. Also, rescaling by the larger of two accumulators is done only when combining a_{med} and a_{sml}. The implementation with these changes is shown in Algorithm 2.

ALGORITHM 2: Blue's algorithm for the 2-norm

```
$a_{sml} = 0$;   $a_{med} = 0$;   $a_{big} = 0$
for $i = 1$ to $n$
    if $|x_i| > t_{big}$
        $a_{big} \leftarrow a_{big} + \left(x_i \cdot s_{big}\right)^2$
    else if $|x_i| < t_{sml}$
        $a_{sml} \leftarrow a_{sml} + (x_i \cdot s_{sml})^2$
    else
        $a_{med} \leftarrow a_{med} + x_i^2$
    end if
end for
if $a_{big} > 0$
    if $a_{med} > 0$ or $isnan(a_{med})$
        $a_{big} \leftarrow a_{big} + \left(a_{med} \cdot s_{big}\right) \cdot s_{big}$
    end if
    Set $w = 1/s_{big}$ and $q = a_{big}$
else if $a_{sml} > 0$
    if $a_{med} > 0$ or $isnan(a_{med})$
        $y_{min} = \min\left(\sqrt{a_{med}}, \sqrt{a_{sml}}/s_{sml}\right)$
        $y_{max} = \max\left(\sqrt{a_{med}}, \sqrt{a_{sml}}/s_{sml}\right)$
        Set $w = 1$ and $q = y_{max}^2\left(1 + (y_{min}/y_{max})^2\right)$
    else
        Set $w = 1/s_{sml}$ and $q = a_{sml}$
    end if
else
    Set $w = 1$ and $q = a_{med}$
end if
```

For xLASSQ, the values of w and q are returned, while for xNRM2 there is one additional step to compute $||x||_2 = w \cdot \sqrt{q}$.

Blue's use of multiple accumulators does not address the accuracy lost by combining elements in the sum of squares with widely differing magnitudes (except, in a limited way, when there are elements on both sides of the cutoff for large or small values). Two entries of the vector may both be in the safe range, but the square of one may overwhelm the other, even though sufficiently many of the smaller elements could affect the sum. This problem is not unique to the 2-norm; it is also present in the dot product, the vector sum, and in higher-level BLAS operations. Since the other BLAS do not rearrange operations to prevent roundoff, we do not attempt it here either.

3.2. Timing Comparisons

Alternatives to Blue's algorithm include the current reference version of xNRM2 on netlib based on LAPACK's xLASSQ, in which the sum of squares is continually rescaled by the largest $|x_i|$ seen so far in a single pass through the vector x, and the 2-pass method for xLASSQ implemented in LAPACK3E, in which the first pass computes $w = \max_i \{|x_i|\}$ and the second pass computes $s = w \cdot \sqrt{\sum_{i=1}^{n} (x_i/w)^2}$. Timing comparisons were conducted on five algorithms for the double precision 2-norm function DNRM2:

1. The original Level 1 BLAS DNRM2 based on Blue's algorithm (Legacy)
2. The current DNRM2 from netlib based on LAPACK's DLASSQ (LAPACK)
3. The 2-pass algorithm for DLASSQ inlined into DNRM2 (2-pass)
4. The new implementation based on Blue's algorithm (Blue)
5. An unsafe algorithm that does no scaling (Unsafe)

The purpose of the timing test is to show that Blue's algorithm can be made competitive with vectorizable algorithms for the 2-norm with a modern implementation. Optimized BLAS, such as Intel MKL [Intel 2015], ATLAS [Whaley et al. 2001], and OpenBLAS, achieve better performance on this algorithm by detecting when scaling can be avoided, implementing kernels in assembly language, and speculatively vectorizing loops that may occasionally contain a branch [Sujon et al. 2013].

The numeric range of the values in the vector x can affect the performance of the algorithms by varying the amount of scaling and branching that is done. The best case for branch prediction is when all the values are in the same range, and the worst case is when they alternate between ranges that are scaled differently. To demonstrate all the features of the latest code, we generated eight different test vectors with values set as follows:

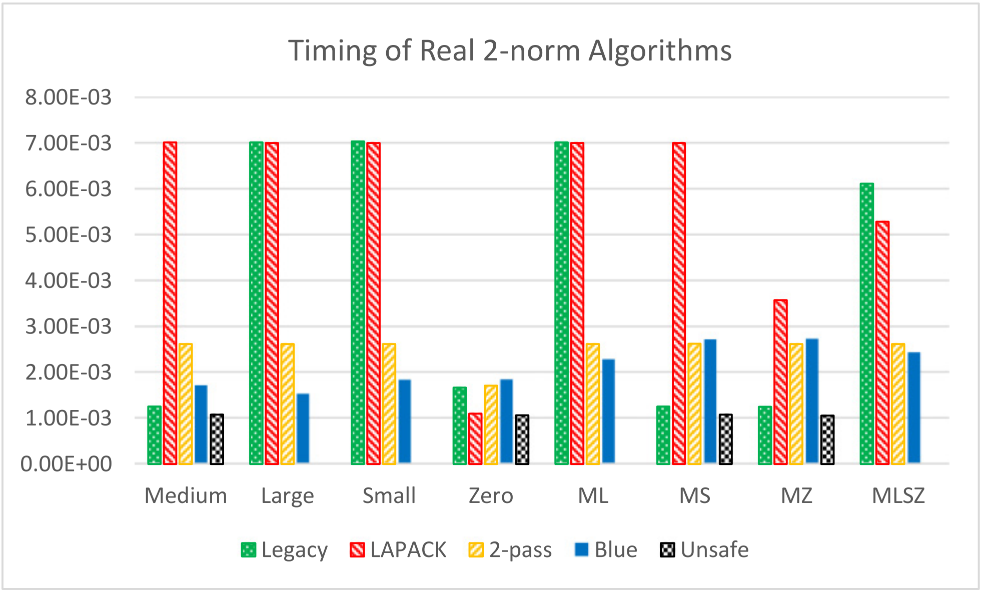
1. Values from the medium range that do not require scaling (Medium)
2. Values from the range that would cause overflow (Large)
3. Values from the range that would cause underflow (Small)
4. All zero (Zero)
5. Alternating medium and large (ML)
6. Alternating medium and small (MS)
7. Alternating medium and zero (MZ)
8. Alternating medium, large, small, and zero (MLSZ)

Tests were conducted both for vectors inside the cache and for vectors constructed as columns of an array that is too large to fit in cache, with similar results, so only the out-of-cache tests will be shown.

Results for the out-of-cache test using one core of a 2.6GHz Intel Xeon E5-2670 processor and the Intel compiler are shown in Figure 1. Each procedure was called $nrep = 100$ times with a vector of length $n = 1{,}000{,}000$, and the time shown is the average time in seconds of a single call. The unsafe algorithm represents the best possible performance but is only able to return a valid result in four of the eight cases. The legacy BLAS and Blue's algorithm are the best performers for a medium vector, which is the case most likely to occur in practice. The LAPACK algorithm is the worst or second-worst performer in all cases except for the zero vector, where it is the best because it tests for zeros. The legacy BLAS performs poorly in several cases where its archaic code logic using computed GOTOs defies the compiler's efforts at branch prediction. The 2-pass algorithm is a steady performer but is beaten by Blue's algorithm in all cases except the zero vector because each element must be accessed twice. For all the performance graphs in this report, Intel Fortran and C compilers version 17.0 were used with the options `-O3 -fp-model precise -xHost`.

4. COMPUTATION OF GIVENS ROTATIONS

A Givens rotation (or plane rotation) is an orthogonal transformation used to introduce zeros selectively in a matrix computation. The original Level 1 BLAS only included subroutines to generate and apply Givens rotations for real data, but the functionality was extended to complex data in LINPACK [Dongarra et al. 1979]. As in the 2-norm function, underflow and overflow must be avoided in intermediate results so that the output parameters are computed accurately.



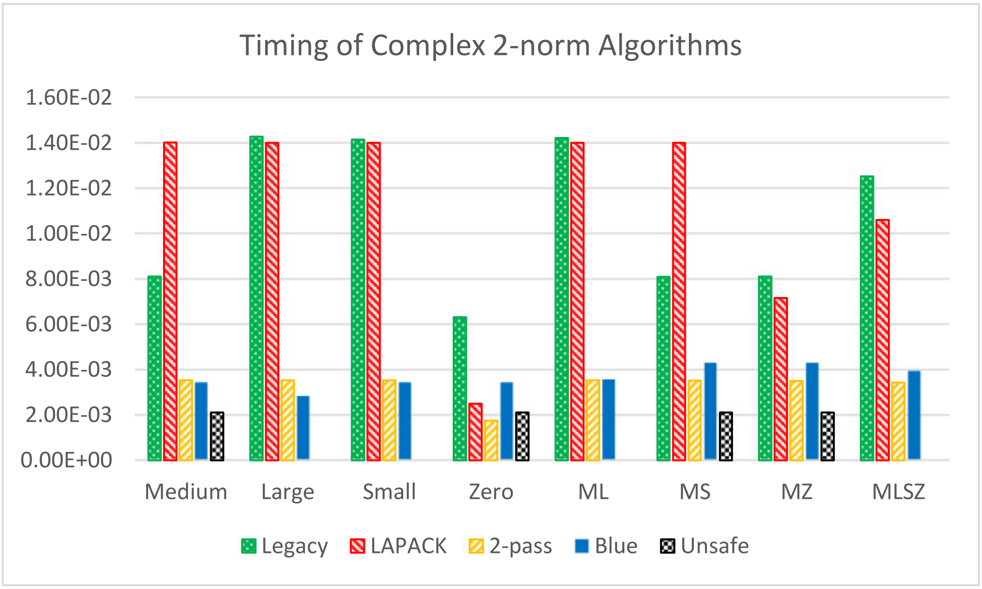


Fig. 1. Out-of-cache timing comparison of DNRM2 and DZNRM2 variants on one core of an Intel Xeon E5-2670 processor (2.6GHz, 20 MB cache), $n = 1{,}000{,}000$; time in seconds (lower is better).

4.1. Real Givens Rotation

In the real case, constructing the rotation matrix reduces to the 2×2 problem of solving the following constrained equation for c and s, given a pair of real input values f and g:

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix} \text{ subject to } c^2 + s^2 = 1.$$

If g is already 0, then there is nothing to do and the computation returns $c = \pm 1$ (depending on the sign desired for r) and $s = 0$. If g is not 0, then the equations can be solved by substitution to give

$$r = \pm\sqrt{f^2 + g^2},\; c = f/r, s = g/r.$$

The choice of sign for r distinguishes the different algorithms for computing the rotation. For consistency with the new BLAS standard [Blackford et al. 2002], we will use the specification from Bindel et al. [2002] and LAPACK3E [Anderson 2002], where r takes the sign of f, making c positive, as shown in Algorithm 3a.

ALGORITHM 3a: Construct Givens rotation given real f and g

```
if $g = 0$
    $c = 1; \quad s = 0; \quad r = f$
else if $f = 0$
    $c = 0; \quad s = \operatorname{sgn}(g); \quad r = |g|$
else
    $d = \sqrt{f^2 + g^2}$
    $c = |f|/d$
    $s = (g \cdot \operatorname{sgn}(f))/d$
    $r = \operatorname{sgn}(f) \cdot d$
end if
```

In this algorithm, for a real number x, $\operatorname{sgn}(x) = \{ {}^{1,\ x \geq 0}_{-1,\ x<0}$

Algorithm 3a has the geometric interpretation of rotating a vector (f, g) in the right half of the $x{-}y$ plane to $(\sqrt{f^2 + g^2}, 0)$ and rotating a vector (f, g) in the left half of the $x - y$ plane to $(-\sqrt{f^2 + g^2}, 0)$. In a sense, vectors that are already pointing in the direction of positive x are rotated to the positive x axis, and vectors that are already pointing in the direction of negative x are rotated to the negative x axis.

As has already been seen in Sections 2 and 3, scaling may be required to avoid underflow or overflow when computing $\sqrt{f^2 + g^2}$. Scaling may also be necessary to ensure that $\sqrt{f^2 + g^2}$ is invertible so that we can replace divisions by this quantity with multiplications by its reciprocal. We will use the safe scaling method from Section 2.3. The computation of a real Givens rotation with this scaling is as shown in Algorithm 3b:

Note that if $|f| \ll |g|$ in Algorithm 3b, $d = \sqrt{f^2 + g^2} \approx \sqrt{g^2} = |g|$ and $|f|/d$ may underflow, making c zero, but this is unavoidable. Similarly, if $|g| \ll |f|$, $d \approx |f|$ and g/d may underflow, making s zero, but this is also unavoidable. It might save a few operations to test for these cases, but the results would be the same.

4.2. Complex Givens Rotation

When f and g are complex, the constrained equation from Section 4.1 becomes

$$\begin{bmatrix} c & s \\ -\bar{s} & c \end{bmatrix} \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix} \text{ subject to } c^2 + s \cdot \bar{s} = 1;$$

where c is real, s and r are complex, and $\bar{s}$ is the complex conjugate of s. Other forms of the complex Givens rotation are possible, for example, making s real or r real instead of c, but in matrix computations it is more efficient to have c or s real because this reduces the cost of applying the rotation to a matrix. For consistency, it is also desired to have the complex algorithm return the same answer as the real algorithm if f and g are real.

ALGORITHM 3b: Construct real Givens rotation with scaling

```
if $g = 0$
    $c = 1; \quad s = 0; \quad r = f$
else if $f = 0$
    $c = 0; \quad s = \mathrm{sgn}(g); \quad r = |g|$
else if (RTMIN $< |f| <$ RTMAX) and (RTMIN $< |g| <$ RTMAX)
    $d = \sqrt{f^2 + g^2}$
    $c = |f|/d$
    $s = (g \cdot sgn(f))/d$
    $r = sgn(f) \cdot d$
else
    $u = \min(\mathrm{SAFMAX}, \max(\mathrm{SAFMIN}, |f|, |g|))$
    $f_s = f/u; \quad g_s = g/u$
    $d = \sqrt{f_s^2 + g_s^2}$
    $c = |f_s|/d$
    $s = (g_s \cdot sgn(f))/d$
    $r = (sgn(f) \cdot d) \cdot u$
end if
```

The complex analog of the algorithm from Section 4.1 is as shown in Algorithm 4a:

ALGORITHM 4a: Construct Givens rotation given complex f and g

```
if $g = 0$
    $c = 1; \quad s = 0; \quad r = f$
else if $f = 0$
    $c = 0; \quad s = \overline{g}/|g|; \quad r = |g|$
else
    $d = \sqrt{|f|^2 + |g|^2}$
    $c = |f|/d$
    $s = \left(\overline{g} \cdot \mathrm{sgn}(f)\right)/d$
    $r = \mathrm{sgn}(f) \cdot d$
end if
```

where, for a complex number z, $|z| = \sqrt{z\overline{z}} = \sqrt{(Re(z))^2 + (Im(z))^2}$ and

$$\mathrm{sgn}(z) = \begin{cases} z/|z|, & z \neq 0 \\ 1, & z = 0 \end{cases}.$$

As before, scaling may be required to avoid underflow or overflow in the computation of $\sqrt{|f|^2 + |g|^2}$ and to ensure that this quantity is invertible. A simple scaling such as

$$u = \min(\mathrm{SAFMAX}, \max(\mathrm{SAFMIN}, |Re(f)|, |Im(f)|, |Re(g)|, |Im(g)|))$$

would fix the sum of squares, but now we must be concerned about underflow when scaling f by a component of g because complex numbers have both a magnitude and a direction, and $\mathrm{sgn}(f)$ may still contribute a direction even when the magnitude of f is very small. If $|f| \ll |g|$, then a different scaling v may need to be applied to f, and the results multiplied by v/u where necessary. On the other hand, underflow in g/u when $|g| \ll |f|$ does not affect the results.

To optimize performance, we would like to avoid computing $|f|$ if possible because it requires another square root. An equivalent formulation employed by Bindel et al.

when f and g are not zero is

$$\begin{aligned}
f_2 &= |f|^2\,;\; g_2 = |g|^2\,; h_2 = f_2 + g_2 \\
d &= \sqrt{f_2 \cdot h_2} \\
p &= 1/d \\
c &= f_2 \cdot p \\
s &= \overline{g} \cdot (f \cdot p) \\
r &= f \cdot (h_2 \cdot p)\,.
\end{aligned}$$

For this formulation to be used, f and g must be well scaled, and even then the algorithm must guard against underflow and overflow when computing $f_2 \cdot h_2$. In the event that this quantity is outside the safe range, the new implementation computes d as $\sqrt{f_2}\sqrt{h_2}$. The construction of a complex Givens rotation with these considerations is shown in Algorithm 4b.

The test "if $f_{max} \ll g_{max}$" is implemented in the code as "if $f_{max}/u <$ RTMIN" using the value of RTMIN from Section 2.2. Although we would like to combine the real quantities $h_2 \cdot p$ and u first in the scaled case before multiplying the complex value f_s, we cannot prevent overflow in $(h_2 \cdot p) \cdot u$ when f is small and g is large, so instead we compute $r = (f_s \cdot (h_2 \cdot p)) \cdot u$ at the cost of one additional multiply.

4.3. Timing Comparisons

Since the generation of a Givens rotation is a scalar procedure, timing comparisons were conducted on the LAPACK routine xLARGV, which is the vector equivalent of the subroutine xLARTG from LAPACK or xROTG from the BLAS or LINPACK. Several methods for constructing Givens rotations were compared:

1. Vector version of xROTG (BLAS)
2. LAPACK3E versions (LAPACK)
3. Optimized vector version of xROTG (ModBLAS)
4. New version described here (New)

The LAPACK3E versions of xLARGV were used instead of LAPACK because the current LAPACK implementations did not pass the timing program's internal checks. It is worth noting that only the new version is able to pass the new rigorous test program at extremes of the floating point range, although all of these versions produced correct answers for the test problems used in the timing program.

To exercise all the different paths through the code, the following different scaling combinations were used for the input values f and g:

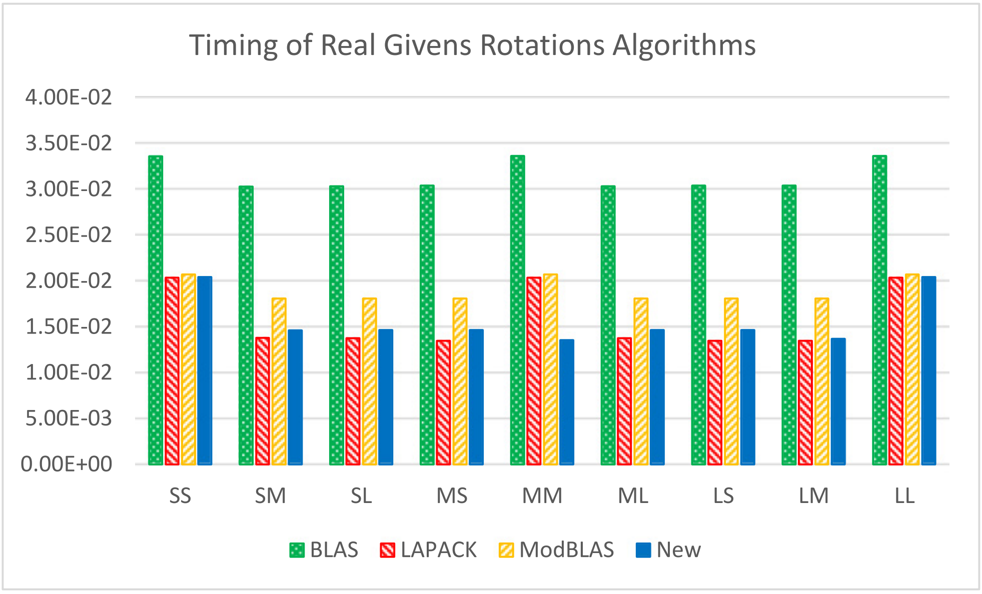
1. f small, g small (SS)
2. f small, g medium (SM)
3. f small, g large (SL)
4. f medium, g small (MS)
5. f medium, g medium (MM)
6. f medium, g large (ML)
7. f large, g small (LS)
8. f large, g medium (LM)
9. f large, g large (LL)

Results for a single core of an Intel Xeon E5-2670 processor using the Intel compiler are shown in Figure 2. A few simple optimizations to the vector version of the BLAS

ALGORITHM 4b: Construct complex Givens rotation with scaling

```
if $g = 0$
    $c = 1; \quad s = 0; \quad r = f$
else if $f = 0$
    $g_{max} = \max(|Re(g)|, |Im(g)|)$
    if (RTMIN $< g_{max} <$ RTMAX)
        $c = 0; \quad s = \overline{g}/|g|; \quad r = |g|$
    else
        $u = \min(\text{SAFMAX}, \max(\text{SAFMIN}, g_{max}))$
        $g_s = g/u$
        $c = 0; \quad s = \overline{g_s}/|g_s|; \quad r = |g_s| \cdot u$
    end if
else
    $f_{max} = \max(|Re(f)|, |Im(f)|)$
    $g_{max} = \max(|Re(g)|, |Im(g)|)$
    if (RTMIN $< f_{max} <$ RTMAX) and (RTMIN $< g_{max} <$ RTMAX) then
        {Use unscaled algorithm}
        $f_2 = |f|^2; \quad g_2 = |g|^2; \quad h_2 = f_2 + g_2$
        if ($f_2 >$ RTMIN) and ($h_2 <$ RTMAX)
            $d = \sqrt{f_2 \cdot h_2}$
        else
            $d = \sqrt{f_2}\sqrt{h_2}$
        end if
        $p = 1/d$
        $c = f_2 \cdot p$
        $s = \overline{g} \cdot (f \cdot p)$
        $r = f \cdot (h_2 \cdot p)$
    else {Use scaled algorithm}
        $u = \min(\text{SAFMAX}, \max(\text{SAFMIN}, f_{max}, g_{max}))$
        $g_s = g/u; \quad g_2 = |g_s|^2$
        if $f_{max} \ll g_{max}$ {Use different scalings for $f$ and $g$}
            $v = \min(\text{SAFMAX}, \max(\text{SAFMIN}, f_{max}))$
            $w = v/u$
            $f_s = f/v; \; f_2 = |f_s|^2; \quad h_2 = f_2 \cdot w^2 + g_2$
        else {Use the same scaling for $f$ and $g$}
            $w = 1$
            $f_s = f/u; \quad f_2 = |f_s|^2; \quad h_2 = f_2 + g_2$
        end if
        if ($f_2 >$ RTMIN) and ($h_2 <$ RTMAX)
            $d = \sqrt{f_2 \cdot h_2}$
        else
            $d = \sqrt{f_2}\sqrt{h_2}$
        end if
        $p = 1/d$
        $c = (f_2 \cdot p) \cdot w$
        $s = \overline{g_s} \cdot (f_s \cdot p)$
        $r = (f_s \cdot (h_2 \cdot p)) \cdot u$
    end if
end if
```

routine xROTG (replacing division by multiplication by the reciprocal and eliminating two calls to CABS in the complex case) made it more competitive and allowed a fairer comparison via the ModBLAS results. The new implementation generally matches the performance of the best BLAS and LAPACK versions in the real case and improves on it in the complex case.



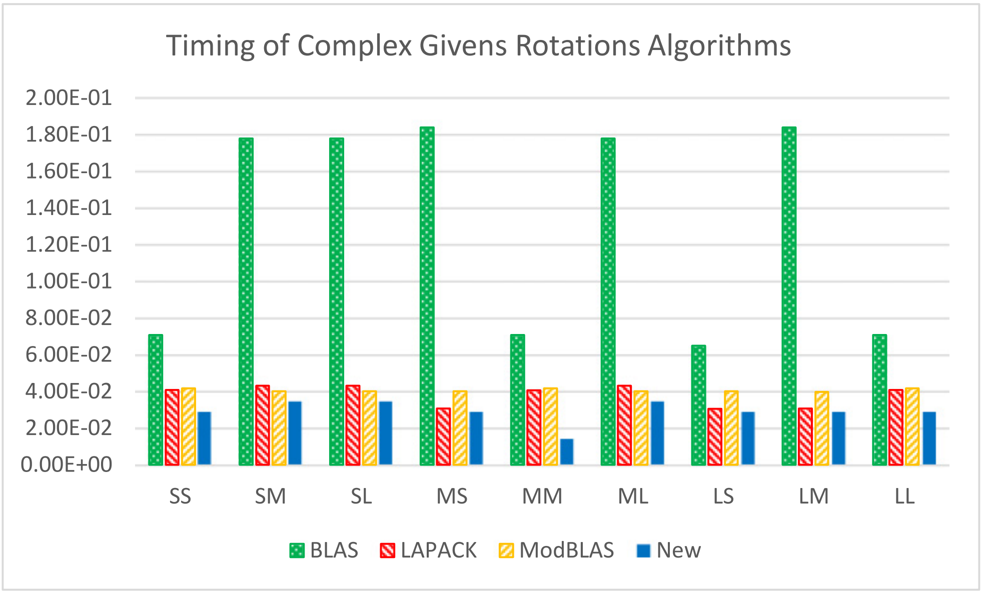


Fig. 2. Out-of-cache timing comparison of DLARGV and ZLARGV variants on one core of an Intel Xeon E5-2670 processor (2.6GHz, 20MB cache); $n = 1{,}000{,}000$; time in seconds (lower is better).

4.4. Floating-Point Model Assumptions

Algorithms 3b and 4b are sufficient to avoid underflow and overflow using some mild assumptions about the floating-point model that are required to prove the following desired relationships between safe and well-scaled numbers.

THEOREM 1. *If r is a real representable number, then with $u = \min(\text{SAFMAX}, \max(\text{SAFMIN}, |r|))$, r/u is well scaled.*

Table II. Calculation of Bound in p-bit Arithmetic for Safe Scaling to Produce a Well-Scaled Number

β	p	t	e_{min}	e_{max}	$(1-e_{min})/2+1$	IEEE format
2	16	11	−13	16	8	binary16
2	32	24	−125	128	64	binary32 (basic)
2	64	53	−1021	1024	512	binary64 (basic)
2	128	113	−16381	16384	8192	binary128 (basic)
10	32	7	−94	97	48	decimal32
10	64	16	−382	385	192	decimal64 (basic)
10	128	34	−6142	6145	3072	decimal128 (basic)

PROOF. We need to show that $|r/u| \leq \sqrt{\text{SAFMAX}}$ even for values of r larger than SAFMAX and that $|r/u| \geq \sqrt{\text{SAFMIN}}$ even for values of r smaller than SAFMIN.

As discussed in Section 2.2, the exponent range is typically fairly symmetric about zero, with a slight bias towards the positive exponents because overflow is a more serious concern than underflow. Assuming a positive exponent bias, the smallest safely invertible number SAFMIN is the smallest model number $\beta^{e_{min}-1}$, and its inverse SAFMAX is $\beta^{1-e_{min}}$. The largest model number is $\beta^{e_{max}}(1-\beta^{-t})$, but for this analysis the upper bound $\beta^{e_{max}}$ is more useful. Hence for any real number r, $|r/u| \leq \beta^{e_{max}+e_{min}-1}$, and for this to be less than $\sqrt{\text{SAFMAX}} = \beta^{(1-e_{min})/2}$, we need $e_{max}+e_{min}-1 \leq (1-e_{min})/2$. The expression on the left side represents the positive exponent bias, while the expression on the right side represents one half of the negative exponent range. The inequality holds easily if the bias is small, as it is in the IEEE programming models. For example, in 32-bit IEEE arithmetic, $e_{max}+e_{min}-1 = 2$, while $(1-e_{min})/2 = 63$. Historically, the exponent ranges have not always been so symmetric; Blue cites the CDC 6000/7000, for which $\beta = 2$ and in double precision $e_{min} = -927$ and $e_{max} = 1070$, but even in this case $e_{max}+e_{min}-1 = 142$, much less than $(1-e_{min})/2 = 464$. For $|r/u|$ to be more than $\sqrt{SAFMAX}$, the hardware designer would have to have allocated 20% or more of the exponent range to the positive bias, which is very unlikely to occur.

At the low end, the ratio is complicated by the possible presence of denormalized numbers, which can admit a value of r much smaller than SAFMIN. Denormalized numbers are denoted by a biased exponent of 0 and correspond to an exponent of e_{min} but with a leading mantissa digit of $m_1 = 0$. The smallest denormalized number is $\beta^{e_{min}-t}$, so using this value for the smallest possible r gives $|r/u| \geq \beta^{-t+1}$. This value is greater than or equal to $\sqrt{\text{SAFMIN}}$ if $-t+1 \geq (e_{min}-1)/2$ or $t-1 \leq (1-e_{min})/2$. The quantity $t-1$ represents the maximum number of shifts that would have to be done to renormalize the result of adding or subtracting two numbers with the same exponent if some of the leading mantissa digits were zero. If $t-1$ were greater than $(1-e_{min})/2$, then adding or subtracting numbers in more than half the negative exponent range could result in a denormalized number due to cancellation. This would be an unacceptably high percentage, so it is unlikely that the hardware designer would ever make the fraction of exponent bits so low. This completes the proof, but it is worth checking the assumptions.

The IEEE 754-2008 standard [IEEE 2008] says that the radix β should be 2 or 10 and that e_{min} should be $1-e_{max}$. The IEEE floating-point model differs from the model described in Section 2 in that the mantissa is normalized to the range $[1, \beta)$, instead of to $[\beta^{-1}, 1)$; adjusting for our model requires adding one to e_{min} and e_{max}, making $e_{min} = 3-e_{max}$ and $e_{max}+e_{min}-1 = 2$. This relationship makes SAFMIN $= \beta^{e_{min}-1}$ and should ensure that $e_{max}+e_{min}-1 \leq (1-e_{min})/2$ as desired. To confirm that $t \leq (1-e_{min})/2+1$ also holds in practice, this bound is calculated for several IEEE floating-point number formats in Table II. In each of the base 2 systems, for p-bit arithmetic, there is 1 bit allocated to the sign, t bits for the mantissa, and k bits for the exponent, but the first mantissa bit is implicit and not stored, so $t + k = p$. Also, $(1 - e_{min})/2 + 1 = 2^{k-2}$ because the exponent range is approximately symmetric with a small positive bias and one reserved value in each half for infinities and NaNs. The allocation of bits is more complicated in the decimal systems, but the ratio of mantissa bits to exponent bits is about the same. All the systems shown satisfy $t \leq (1 - e_{min})/2 + 1$, except the system with $t = 11, k = 5$, and $p = 16$, which corresponds to IEEE "half" precision, an interchange format that by implication is not intended for arithmetic use. We see that $|r/u| \geq \sqrt{\text{SAFMIN}}$ holds with room to spare for all the standard floating-point number systems with 32 bits of precision or more.

Theorem 2. *If z is a complex representable number, then with $u = \min(\text{SAFMAX}, \max(\text{SAFMIN}, |Re(z)|, |Im(z)|))$, z/u is well scaled.*

Proof. Recall from Section 2.2 that a vector of length 2 is well scaled if the sum of squares of its components can be computed by an unscaled algorithm without underflow or overflow. If $z = (0, 0)$, then $u = \text{SAFMIN}$ and $z/u = (0,0)$, which is trivially well scaled. If z is nonzero, then at least one of $Re(z)$ or $Im(z)$ is nonzero. Let $z_{min} = \min(|Re(z)|, |Im(z)|)$ and $z_{max} = \max(|Re(z)|, |Im(z)|)$, and suppose $\text{SAFMIN} \leq z_{max} \leq \text{SAFMAX}$, so that $u = z_{max}$. Then $|z/u|^2 = 1 + (z_{min}/z_{max})^2$ and $1 \leq 1 + (z_{min}/z_{max})^2 \leq 2$, so z/u is well scaled for this case.

Now suppose $|z_{max}| > \text{SAFMAX}$, so $u = \text{SAFMAX}$. In this case, $|z/u|^2 = (z_{min}/u)^2 + (z_{max}/u)^2 \leq 2 \cdot |z_{max}/\text{SAFMAX}|$, so for z/u to be well scaled, we require $|z_{max}/\text{SAFMAX}| \leq \sqrt{\text{SAFMAX}/2}$. As in the proof of Theorem 1, the left side is at most $\beta^{e_{max}+e_{min}-1}$ and the right side is $\beta^{(1-e_{min})/2}$ divided by $\sqrt{2}$, or $\beta^{-1/2}$ when $\beta = 2$, which will hold (for $\beta = 2$) if $e_{max} + e_{min} - 1 \leq -e_{min}/2$. The left side represents the positive exponent bias and is always 2 in IEEE arithmetic, while the right side is half the negative exponent range, an exponential historically much larger than the positive exponent bias.

If $|z_{max}| < SAFMIN$, then $u = \text{SAFMIN}$. In this case, $|z/u|^2 = (z_{min}/u)^2 + (z_{max}/u)^2 \geq (z_{max}/u)^2 \geq \text{SAFMIN}$ from Theorem 1. □

Corollary 1. *If a complex representable number z is well scaled, then $|z|$ and $z/|z|$ are well scaled.*

Theorem 3. *The product of two well-scaled numbers is a representable number.*

Theorem 4. *The product of a well-scaled number and a number of unit norm is well scaled.*

Theorem 5. *The product of a safe number and a number of unit norm is safe.*

The proofs of Theorems 3–5 follow directly from the definitions and the norm property $|z_1 \cdot z_2| = |z_1| \cdot |z_2|$.

4.5. Proof of Safety

We will say that an algorithm is safe if its output results do not overflow or underflow unless overflow or underflow cannot be avoided. Overflow or underflow cannot be avoided in every circumstance, and in these cases the proper result is to return plus or minus infinity or zero. We now examine Algorithm 4b to see if it is safe. The proof of Algorithm 3b follows as a special case.

If g is zero, the algorithm does no computation, simply returning with $r = f$. If f is zero, g is a representable number, and $\text{RTMIN} < \max(|Re(g)|, |Im(g)|) < \text{RTMAX}$, then g is well scaled and $|g|$ is both well scaled and safe, so s and r are both well scaled. If g requires scaling, the scaling constant u makes g well scaled by Theorem 2, hence s is well scaled, and the quantity r is a product of a well-scaled number and a safe number. We cannot conclude anything about r, and it may overflow, but if it does, overflow was unavoidable. If g contains a NaN, then g_{max} may or may not be NaN; the IEEE standard is ambiguous in this case. But regardless of what scaling is computed, at least one component of g_s will be NaN, so $|g_s|$ will be NaN, making s and r both NaN. If g contains an infinite value and no NaNs, then $|g_s|$ will be infinite, s will contain NaN, and r will be infinite.

If f and g are both nonzero and well scaled, then $f_2 = |f|^2, g_2 = |g|^2$, and $h_2 = f_2+g_2$ are all computable without underflow or overflow. Since $f_2 \le f_2 + g_2 = h_2$, the conditions $f_2 >$ RTMIN and $h_2 <$ RTMAX are equivalent to RTMIN $< f_2 <$ RTMAX and RTMIN $< h_2 <$ RTMAX, hence SAFMIN $< f_2 \cdot h_2 <$ SAFMAX, which guarantees that the product is safe and its square root is well scaled, so a single square root can be used. If the product is not safe, the conditions on f and g still ensure that $|f|$ and $\sqrt{|f|^2 + |g|^2}$ are well scaled and that $d = \sqrt{f_2}\sqrt{h_2}$ is safe, hence its inverse p is also safe. The cosine $c = f_2 \cdot p$ is the product of two real representable numbers and is equivalent to $c = |f|/\sqrt{|f|^2 + |g|^2}$, which is always positive and between 0 and 1 inclusive. The sine s is the product of a well-scaled number $\overline{g}$ and the product $f \cdot p = f/(|f|\sqrt{|f|^2 + |g|^2})$, which is well scaled as the product of a number of unit norm and a well-scaled number, hence s is representable. The quantity r is the product of a well-scaled number f and the product $h_2 \cdot p = (|f|^2 + |g|^2)/(|f|\sqrt{|f|^2 + |g|^2}) = \sqrt{|f|^2 + |g|^2}/|f|$. This expression must be at least 1 because $|f| > 0$. If $|f| \ge |g|$, then $\sqrt{|f|^2 + |g|^2} \le \sqrt{2|f|^2}$ and $\sqrt{|f|^2 + |g|^2}/|f| \le \sqrt{2}$. If $|f| < |g|$, then $\sqrt{|f|^2 + |g|^2} \le \sqrt{2|g|^2}$ and $\sqrt{|f|^2 + |g|^2}/|f| \le \sqrt{2}|g|/|f| \le \sqrt{2}$ RTMAX/RTMIN $= \sqrt{2}(\text{RTMAX})^2 = \sqrt{2}(\text{SAFMAX} \cdot \text{ULP})$, showing that $h_2 \cdot p$ is safe. Thus r is the product of two representable numbers, which is as safe as we can make it.

If f and g are both nonzero but not well scaled, then both are scaled to form f_s and g_s. The scaling constant $u = \min(\text{SAFMAX}, \max(\text{SAFMIN}, f_{max}, g_{max}))$ will make one of f_s or g_s well scaled by Theorem 5. Regardless of whether f or g is larger in magnitude, g is scaled by u. If $f_{max} \ll g_{max}$, an alternate scaling constant $v = \min(\text{SAFMAX}, \max(\text{SAFMIN}, f_{max}))$ is applied to f; this makes f_s well scaled and is important to prevent underflow in f_s. However, if scaling by u will not cause underflow, f is also scaled by u. Then $f_2 = |f_s|^2$, $g_2 = |g_s|^2$, and either $h_2 = f_2 \cdot (v/u)^2 + g_2$ or $h_2 = f_2 + g_2$, and the rest of the analysis proceeds as in the unscaled case with scaling factors added to the formulas for c and r. The constant $w = v/u$ may not be safe with respect to underflow if the magnitude of g is much larger than that of f, but in this case, referring back to Algorithm 4a, underflow is unavoidable in $c = |f|/\sqrt{|f|^2 + |g|^2} \approx |f|/|g|$.

If f is nonzero and f or g contains a NaN, then at least one component of the sum of squares in h_2 will be NaN, so d will be NaN, p will be NaN, and c, s, and r will all be NaN, as desired. If f or g contain an infinite value, then at least one component of the sum of squares in d will be infinite, so c and s will be 0 or NaN and r will be infinite. We could save a few operations by testing for $g_{max} \ll f_{max}$, but it would not change the results.

5. OTHER IMPROVEMENTS TO THE BLAS

Other than the scaling issues when computing the 2-norm of a vector or constructing Givens rotations, the primary concern with the original BLAS is making them consistent with successor packages, such as the matrix-vector (Level 2) BLAS

[Dongarra et al. 1988], the matrix-matrix (Level 3) BLAS [Dongarra et al. 1990], LINPACK [Dongarra et al. 1979], and LAPACK [Anderson et al. 1999]. As the BLAS have evolved, the interfaces have standardized on allowing the vector increment to be positive or negative, a feature only partially supported in the Level 1 BLAS. The reference versions of the Level 2 and 3 BLAS are very straightforward, leaving any optimizations such as was attempted with manual unrolling in the original BLAS to the developers of numerical libraries. The test programs for the Level 2 and 3 BLAS and LAPACK are also driven by a data file, allowing them to test a user-selectable range of problem sizes in order to exercise all paths in an implementation optimized by a library developer for a particular architecture.

5.1. Naming Conventions

The names of the Level 1 Basic Linear Algebra Subprograms take the form

$< \text{datatype(s)} >< \text{function} >$

where the standard data types are

I	Integer
S	Single precision real
D	Double precision real
C	Single precision complex
Z	Double precision complex

Functions begin with the character matching their return data type, while subroutines begin with the character matching the data type of their vector arguments. When the interface contains both real and complex arguments, two of these data types may be used in the subroutine name. For example, the 2-norm of a complex vector in double precision is DZNRM2, indicating that the return type is double precision real and the vector argument is double precision complex, and there is both a ZDSCAL and ZSCAL for scaling a double precision complex vector, the first scaled by a double precision real scalar and the second by a double precision complex scalar.

The full list of BLAS modified by this work is as follows:

Data types	Name	Description
S,D,C,Z	IxAMAX	Maximum element in absolute value
S,D,SC,DZ	xASUM	1-norm of a vector
S,D,C,Z	xAXPY	Sum of two vectors: $y \leftarrow ax + y$
S,D,C,Z	xCOPY	Copy a vector: $y \leftarrow x$
S,D	xDOT	Real dot product: $x^T y$
C,Z	xDOTC	Complex dot product, conjugated: $\overline{x}^T y$
C,Z	xDOTU	Complex dot product, unconjugated: $x^T y$
S,D,SC,DZ	xNRM2	2-norm of a vector
S,D,C,Z,CS,ZD	xROT	Apply a Givens rotation
S,D,C,Z	xROTG	Compute a Givens rotation
S,D,C,Z,CS,ZD	xSCAL	Scale a vector: $x \leftarrow ax$
S,D,C,Z	xSWAP	Swap two vectors

The functionality of the routines xAXPY, xCOPY, xDOT, xROT, and xSWAP is unchanged from the original BLAS, but updated versions of these routines are provided for consistency in programming style. The functions C/Z ROTG, C/Z ROT, and CS/ZD ROT are BLAS-like extensions provided with LINPACK and/or LAPACK.

New implementations are also provided for the following LAPACK auxiliary routines:

Data types	**Name**	**Description**
S,D,C,Z	xLARTG	Compute a Givens rotation
S,D,C,Z	xLASSQ	Compute a scaled sum of squares

The KIND is parameterized in each of these routines, so additional variations, such as 128-bit real or complex versions, could easily be constructed.

5.2. General Vector Increment

The BLAS and LAPACK subroutines that include a vector argument typically also include a vector increment *INCX* describing the spacing between consecutive elements of the n-element vector x. Following a convention that goes back to the original BLAS article, if $INCX > 0$ then the ith element of x is stored in $X(1 + (i - 1) * INCX)$, while if $INCX < 0$ then the ith element of x is stored in $X(1 - (n - i) * INCX)$ for $1 \leq i \leq n$. This method of indexing when $INCX < 0$ avoids negative indices in the array X, in compliance with the Fortran standard of the time in which these subroutines were developed.

Oddly enough, the implementors of the Level 1 BLAS did not follow this convention for subroutines operating on a single vector. For these subroutines, the authors said, “Only positive values of INCX are allowed.” However, the implementation did not check for a negative increment and behaved irrationally if a negative increment was supplied, in some cases iterating from 1 to $1 + (n - 1) * INCX$ in steps of INCX, that is, from 1 to $-n + 2$ if INCX were -1, referencing elements of x with a negative index in violation of the Fortran standard of that time. In the 1990s, the reference BLAS on netlib [Dongarra and Grosse 1987] were modified to quietly return without an error condition if $INCX < 0$ in those subprograms that purported not to allow negative increments. However, some vendor libraries, such as the Cray Scientific Library [Cray Research 1993], the IBM Engineering and Scientific Subroutine Library [IBM 2012], and the Intel Math Kernel Library [Intel 2015], had already implemented negative increments for single-vector subroutines the same as for two-vector subroutines. The new model implementation standardizes this extension. In the discussion that follows, variables from the Fortran subroutines are referred to by their names in lowercase letters, that is, x and $incx$, rather than X and $INCX$, to match the style of the new model implementation.

The algorithms for the 1-norm xASUM, the infinity norm IxAMAX, and the vector scaling xSCAL are straightforward to implement with provision for a general increment that can be positive, negative, or zero (a general vector increment was also added to xNRM2, described in Section 3). To minimize loop overhead, a special case is provided for each of these operations when $incx = 1$, the most commonly occurring case. For example, the main loop for $incx = 1$ in xSCAL is

```
do i = 1, n
   x(i) = alpha*x(i)
end do
```

while the loop for $incx \neq 1$ is

```
ix = 1
if( incx < 0 ) ix = 1 - (n-1)*incx
do i = 1, n
   x(ix) = alpha*x(ix)
   ix = ix + incx
end do
```

Note that the starting point for the vector x is moved to the end of the array, and the array is traversed in reverse order when $incx < 0$.

Negative increments are rarely used in linear algebra software because the Level 2 and 3 BLAS allow a general stride in only one dimension of a two-dimensional (2D) array, and that stride must be positive. However, predecessors to the BLAS, such as matrix-vector multiply (MXVA) and matrix-matrix multiply (MXMA) from the Cray Scientific Library [Cray Research 1993], supported both a column increment *iac* and a row increment *iar* for a 2D array A, and the increments could be positive or negative. This approach allows a smaller number of kernels to be developed in a low-level language [Sheikh et al. 1992]. For example, it is only necessary to develop a lower triangular system solve, because if a two-dimensional matrix A has $iac = 1$ and $iar = lda$, the solution vector x has stride $incx = 1$, and the right-hand side vector y has stride $incy = 1$, an upper triangular system $Ax = y$ can be solved by the lower triangular kernel with $iac = -1$, $iar = -lda$, $incx = -1$, and $incy = -1$. It is unclear if the original BLAS authors had this in mind.

An increment of 0 would mean that the vector is not a vector at all but a scalar. Zero increments are flagged as an error in LAPACK and the Level 2 BLAS, in part because allowing them in an output vector x would inhibit vectorization or lead to unexpected results. For example, a matrix-vector multiply $y = Ax$ with $incy = 0$ would reduce to a dot product of the last row of A with the vector x. In the Level 1 BLAS, there is no provision for error handling, so the new model implementation allows zero increments, and the test program validates them, but their use is strongly discouraged.

5.3. Special Handling of NaNs

In general, when NaNs are encountered on input, it is desirable to have them propagate to the output result. The propagation will occur naturally in floating-point combinations as are found in xAXPY, xROT, and xSCAL, but care must be taken to ensure that optimizations to the norm and dot product routines do not optimize away the NaNs. An adjustment must also be considered for the function IxAMAX to find the index of the maximum element in absolute value.

The current reference version of IxAMAX initializes the maximum value to the absolute value of the first element of the vector x and then scans elements 2 through n to see if any are larger. If the first element is NaN, all comparisons to it will fail, so IxAMAX will return 1, while if the first element is not NaN, all comparisons against subsequent NaN values will fail, and the index returned will be of the largest non-NaN value. To fix this inconsistency, we initialize the maximum value to -1.0 and the index to 1 and scan elements 1 through n for a larger element. If all entries are NaN, the function returns 1 as before; otherwise, it will return the index of the largest non-NaN value. This behavior is consistent with how NaNs are handled in the Fortran intrinsic functions MAXLOC and MAXVAL and by statistical packages such as Matlab and R.

5.4. Test Program

New test code for the Level 1 BLAS has been developed, replacing the Level 1 BLAS test program, which contained only pre-computed problems of dimension 5 or less. The new test program is driven by an input file in the style of the Level 2 or Level 3 BLAS test suites, from which the problem sizes, values of the scalar parameters, and test paths can be selected. Test inputs to xASUM, xNRM2, xROTG/xLARTG, and xLASSQ are set to a range of positive and negative values, including values near zero, underflow, and overflow to exercise all the scaling pitfalls. NaN values are also used in the test programs for IxAMAX, IxASUM, xDOT, xNRM2, xLARTG, and xLASSQ to verify the expected propagation of NaNs. For xAMAX, xAXPY, xCOPY, xROT, xSCAL, and xSWAP, the correct answer is known or readily computed, and results should agree exactly. For xASUM, xDOT, xNRM2, and xLASSQ, test vectors are constructed with values between -1 and 1 and scaled to the desired range, and the expected result is calculated by a straightforward algorithm on the vector before scaling. Roundoff errors do occur and grow like $O(n)$, so the resulting test ratios must be scaled by n; for example, the test ratio for the relative error in the computed value t_{com} for SNRM2 compared to the expected value t_{exp} is

$$\frac{|t_{com} - t_{exp}|}{|t_{exp}| \cdot n \cdot \varepsilon},$$

and this value should be $O(1)$. For xROTG/xLARTG, the equation being solved (in the real case) is

$$\begin{bmatrix} c & s \\ -s & c \end{bmatrix} \begin{bmatrix} f \\ g \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix} \text{ subject to } c^2 + s^2 = 1,$$

and the constructed values of c, s, and r are verified by checking that

(1) $c \cdot f + s \cdot g = r$
(2) $-s \cdot f + c \cdot g = 0$
(3) $c^2 + s^2 = 1.$
(4) Sign of c is correct ($c \geq 0$ in xLARTG)

Validation of results includes an examination of all memory locations that should not be modified by the subroutine or function call to confirm that their values have not changed. Tests were conducted using the Intel, Gnu, and PGI compilers on a single node of an IBM iDataPlex system having an Intel Xeon processor. All program units compiled without warnings, and all tests passed with the new model implementation.

Previous versions of the BLAS and LAPACK routines tested here fail the newly constructed tests in both expected and unexpected ways. The legacy BLAS routines xASUM, xNRM2, IxAMAX, and xSCAL fail tests with negative increments because they did not support them. The LAPACK versions of xLASSQ fail tests with input values of $w = 1.0$ and $q = 0.0$ and a vector scaled near zero because they do not do any scaling in this case. The reference versions of xNRM2 and xLASSQ fail tests with infinite values by returning NaN due to the unsafe scaling employed. The reference version of IxAMAX fails tests with $x(1) = \text{NaN}$ by returning 1, whereas the test program expects the index of the first non-NaN if one exists as described in Section 5.3. The legacy BLAS and LAPACK routines to compute Givens rotations are not consistent between the real and complex versions when provided with real inputs because this feature was not part of their design, and they go into an infinite loop for certain badly scaled input values. The routines to compute Givens rotations described in Bindel et al. [2002] produce wrong answers or unexpected NaNs in the complex case due to overflow in the intermediate results.

The design of the new test code and model implementation allows for easy extension to additional data types beyond the four provided ones. As in LAPACK3E, the KIND and all common constants such as zero and one are coded as Fortran parameters and included via a module, rather than hard-coded in every program unit as in LAPACK. Floating-point model parameters such as epsilon, SAFMIN, and SAFMAX are also specified as parameters in the constants module, rather than using an inquiry function with the cost of a function call on every use as in LAPACK. Generic intrinsic functions are used wherever possible, and the kind is added to the CMPLX intrinsic function for portability. Most importantly, the new implementation is compact and readable in the style of the BLAS or of LINPACK, allowing it to serve as a model for robust software design.

6. CONCLUSION

This article describes algorithmic improvements made in reimplementing the Level 1 BLAS and related subroutines from LAPACK. Performance tests show that the new implementations are as fast or faster than the existing reference implementations, particularly for the 2-norm and for any subroutine requiring calls to the LAPACK inquiry function. A new comprehensive test program demonstrates correctness even for extreme values in the floating-point model and exposes the limitations of the reference versions that are hereby replaced.

The modular design of the software and test procedures allows for easy extensions to other data types, such as 128-bit precision. Defining all program constants in a module is both good programming practice and a performance improvement over the use of an inquiry function. Safe scaling as done here is easy to describe and leads to more straightforward implementations of basic algorithms from LAPACK and related software. In addition, a common source for all real data types or all complex data types could be achieved by selecting the kind at compile time as in LAPACK3E and using generic function names in place of type-specific names. Generic interfaces have the advantage of supporting compile-time verification of the number and type of arguments, but a fully general interface supporting all array reshaping leads to an exponential explosion of interfaces, and if more than one subroutine is included in the module, dependencies are created to other possibly unwanted library routines.

Like the previous BLAS, this implementation does not compensate for roundoff errors that occur when adding a sequence of numbers. With the exception of Blue's algorithm for the 2-norm, values are added from 1 to n, and roundoff errors that occur when adding two values of widely differing magnitudes are dropped. Blue's algorithm does some grouping of values by magnitude, but roundoff errors within those groups are also dropped. The test program constructs test cases for the worst case of roundoff error, which is $O(n)$ for the dot product, 1-norm, 2-norm, and sum of squares and, consequently, has to scale the resulting test ratios by $1/n$. Introducing some form of compensated summation as described, for example, in Higham [1996], would allow this test to be more strict.

APPENDIX: CONSTANTS MODULES FOR 32-BIT AND 64-BIT IEEE ARITHMETIC

```
module LA_CONSTANTS32
!
! -- BLAS/LAPACK module --
! May 06, 2016
!
! Standard constants
!
  integer, parameter :: wp = 4
  real(wp), parameter :: zero = 0.0_wp
  real(wp), parameter :: half = 0.5_wp
  real(wp), parameter :: one = 1.0_wp
  real(wp), parameter :: two = 2.0_wp
  real(wp), parameter :: three = 3.0_wp
  real(wp), parameter :: four = 4.0_wp
  real(wp), parameter :: eight = 8.0_wp
  real(wp), parameter :: ten = 10.0_wp
  complex(wp), parameter :: czero = ( 0.0_wp, 0.0_wp )
  complex(wp), parameter :: chalf = ( 0.5_wp, 0.0_wp )
  complex(wp), parameter :: cone = ( 1.0_wp, 0.0_wp )
  character*1, parameter :: sprefix = 'S'
  character*1, parameter :: cprefix = 'C'
!
! Model parameters
!
  real(wp), parameter :: eps = 0.5960464478E-07_wp
  real(wp), parameter :: ulp = 0.1192092896E-06_wp
  real(wp), parameter :: safmin = 0.1175494351E-37_wp
  real(wp), parameter :: safmax = 0.8507059173E+38_wp
  real(wp), parameter :: smlnum = 0.9860761315E-31_wp
  real(wp), parameter :: bignum = 0.1014120480E+32_wp
  real(wp), parameter :: rtmin = 0.3140184864E-15_wp
  real(wp), parameter :: rtmax = 0.3184525782E+16_wp
!
! Blue's scaling constants
!
  real(wp), parameter :: tsml = 0.1084202172E-18_wp
  real(wp), parameter :: tbig = 0.4503599627E+16_wp
  real(wp), parameter :: ssml = 0.3777893186E+23_wp
  real(wp), parameter :: sbig = 0.1323488980E-22_wp
end module LA_CONSTANTS32

module LA_CONSTANTS
!
! -- BLAS/LAPACK module --
!    May 06, 2016
!
! Standard constants
!
  integer, parameter :: wp = 8
  real(wp), parameter :: zero = 0.0_wp
  real(wp), parameter :: half = 0.5_wp
  real(wp), parameter :: one = 1.0_wp
  real(wp), parameter :: two = 2.0_wp
  real(wp), parameter :: three = 3.0_wp
  real(wp), parameter :: four = 4.0_wp
  real(wp), parameter :: eight = 8.0_wp
  real(wp), parameter :: ten = 10.0_wp
  complex(wp), parameter :: czero = ( 0.0_wp, 0.0_wp )
  complex(wp), parameter :: chalf = ( 0.5_wp, 0.0_wp )
  complex(wp), parameter :: cone = ( 1.0_wp, 0.0_wp )
  character*1, parameter :: sprefix = 'D'
  character*1, parameter :: cprefix = 'Z'
!
! Model parameters
!
  real(wp), parameter :: eps = 0.11102230246251565404E-015_wp
  real(wp), parameter :: ulp = 0.22204460492503130808E-015_wp
  real(wp), parameter :: safmin = 0.22250738585072013831E-307_wp
  real(wp), parameter :: safmax = 0.44942328371557897693E+308_wp
  real(wp), parameter :: smlnum = 0.10020841800044863890E-291_wp
  real(wp), parameter :: bignum = 0.99792015476735990583E+292_wp
  real(wp), parameter :: rtmin = 0.10010415475915504622E-145_wp
  real(wp), parameter :: rtmax = 0.99895953610111751404E+146_wp
!
! Blue's scaling constants
!
  real(wp), parameter :: tsml = 0.14916681462400413487E-153_wp
  real(wp), parameter :: tbig = 0.19979190722022350281E+147_wp
  real(wp), parameter :: ssml = 0.44989137945431963828E+162_wp
  real(wp), parameter :: sbig = 0.11113793747425387417E-161_wp
end module LA_CONSTANTS
```

ACKNOWLEDGMENTS

I am grateful to Damian McGuckin of Pacific ESI for his draft article [McGuckin 2010] that suggested revisiting Blue's algorithm for the sum of squares. Comments from Jim Demmel, Fabian Lischka, and the referees helped to improve the software implementations and the presentation.

REFERENCES

E. Anderson. 2002. *LAPACK3E—A Fortran 90-enhanced Version of LAPACK*. UT-CS-02–497 (LAPACK Working Note 158). University of Tennessee, Knoxville.

E. Anderson and M. Fahey. 1997. *Performance Improvements to LAPACK for the Cray Scientific Library*. UT-CS-97-359 (LAPACK Working Note 126). University of Tennessee, Knoxville.

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

D. Bindel, J. Demmel, W. Kahan, and O. Marques. 2002, On computing Givens rotations reliably and efficiently. *ACM Trans. Math. Soft.* 28, 2 (Jun. 2002), 206–238.

L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Soft.* 28, 2 (Jun. 2002), 135–151.

J. L. Blue. 1978. A portable Fortran program to find the Euclidean norm of a vector. *ACM Trans. Math. Soft.* 4, 1 (Mar. 1978), 15–23.

W. J. Cody. 1988. MACHAR: A subroutine to dynamically determine machine parameters. *ACM Trans. Math. Soft.* 14, 4 (Dec. 1988), 303–311.

Cray Research Inc. 1993. *Scientific Libraries Reference Manual*. SR-2081 8.0.

J. Demmel. 1997. *Applied Numerical Linear Algebra*. SIAM, Philadelphia.

J. Demmel. 2002. Software for Accurate and Efficient Givens Rotations. Retrieved from http://www.cs.berkeley.edu/~demmel/Givens.

J. J. Dongarra. 1980. Fortran BLAS Timing. ANL-80-24 (LINPACK Working Note #3). Argonne National Laboratory, Argonne, IL.

J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. 1979. *LINPACK Users' Guide*. SIAM, Philadelphia.

J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. 1990. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.* 16, 1 (Mar. 1990), 1–17.

J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. 1988. An extended set of Fortran basic linear algebra subprograms. *ACM Trans. Math. Soft.* 14, 1 (Mar. 1988), 1–17.

J. J. Dongarra and E. Grosse. 1987. Distribution of mathematical software via electronic mail. *Commun. ACM* 30, 5 (May 1987), 403–407.

P. A. Fox, A. D. Hall, and N. L. Schryer. 1978. Algorithm 528: Framework for a portable library. *ACM Trans. Math. Soft.* 4, 2 (Jun. 1978), 177–188.

R. J. Hanson and T. Hopkins. 2004. Algorithm 830: Another visit with standard and modified Givens transformations and a remark on algorithm 539. *ACM Trans. Math. Soft.* 30, 1 (Mar. 2004), 86–94.

N. J. Higham. 1996. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia.

T. Hopkins. 1997. Restructuring the BLAS level 1 routine for computing the modified Givens transformation. *ACM SIGNUM Newslett*. 32, 4 (Oct. 1997), 2–14.

IBM. 2012. *IBM Engineering and Scientific Subroutine Library (ESSL) Guide and Reference*, Version 5.1 Release 1.

IEEE. 2008. IEEE Standard for Floating-Point Arithmetic. Institute for Electrical and Electronics Engineers Inc., New York, NY.

Intel. 2015. *Intel Math Kernel Library Reference Manual*.

C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.* 5, 3 (Sep. 1979), 308–323.

D. McGuckin. 2010. *N-Dimensional Euclidean Norms*. (personal communication).

Q. Sheikh, P. Vu, C. Yang, and M. Merchant. 1992. Implementation of the level 2 and 3 BLAS on the CRAY Y-MP and CRAY-2. *J. Supercomput.* 5 (1992), 291–305.

M. H. Sujon, R. C. Whaley, and Q. Yi. 2013. Vectorization past dependent branches through speculation. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, 353–362.

R. C. Whaley, A. Petitet, and J. J. Dongarra. 2001. Automated empirical optimization of software and the ATLAS project. *Parallel Comput*. 27, 1–2 (2001), 3–35.

Received June 2015; revised March 2017; accepted March 2017