Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm

RICHARD J. HANSON, Retired
TIM HOPKINS, University of Kent, UK

We propose a set of new Fortran reference implementations, based on an algorithm proposed by Kahan, for the Level 1 BLAS routines **NRM2* that compute the Euclidean norm of a real or complex input vector. The principal advantage of these routines over the current offerings is that, rather than losing accuracy as the length of the vector increases, they generate results that are accurate to almost machine precision for vectors of length $N < N_{max}$ where N_{max} depends upon the precision of the floating point arithmetic being used. In addition we make use of intrinsic modules, introduced in the latest Fortran standards, to detect occurrences of non-finite numbers in the input data and return suitable values as well as setting IEEE floating point status flags as appropriate. A set of C interface routines is also provided to allow simple, portable access to the new routines.

To improve execution speed, we advocate a hybrid algorithm; a simple loop is used first and, only if IEEE floating point exception flags signal, do we fall back on Kahan's algorithm. Since most input vectors are 'easy', i.e., they do not require the sophistication of Kahan's algorithm, the simple loop improves performance while the use of compensated summation ensures high accuracy.

We also report on a comprehensive suite of test problems that has been developed to test both our new implementation and existing codes for both accuracy and the appropriate settings of the IEEE arithmetic status flags.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Computer Arithmetic, Numerical Linear Algebra

General Terms: Design, Algorithms, Performance, Portability, Reliability

Additional Key Words and Phrases: Euclidean Norms, Exception Handling, Fortran, C, Language Interoperability, Compensated Summation, Reliability, Accuracy

ACM Reference Format:
Hanson, R., Hopkins, T., Remark on Algorithm 539: Carefully Computing the Euclidean Norm, *ACM Trans. Math. Softw.* vv, nn, Article aa (January 201y), 23 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The Level 1 BLAS [Lawson et al. 1979b] is an ensemble algorithm designed to perform a number of low-level computations on rank-1 assumed size arrays. The package was published prior to the availability of the IEEE 754 floating point standard [P754 1985] and the language standards known as Fortran 90, 95, 2003 and 2008 [ISO/IEC 1991; 1997; 2004; 2011]. A Fortran reference version was made available as part of Algorithm 539 [Lawson et al. 1979a].

The software associated with Algorithm 539 was written in Fortran 66 [ANSI 1966]. The introduction into Fortran 2003 of the intrinsic modules *IEEE_ARITHMETIC* and *IEEE_EXCEPTIONS* gives the programmer standard conforming access to much of the

Author's addresses: T. Hopkins, Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK; email: t.r.hopkins@kent.ac.uk.

DOI: http://dx.doi.org/10.1145/0000000.0000000

functionality of IEEE floating point arithmetic. This provides the opportunity to improve the Level 1 BLAS functions, **NRM2*, that compute the l_2 norm of a vector. (*SNRM2* and *SCNRM2* are the specific names for single-precision real and complex input vectors respectively; the corresponding names for double precision are *DNRM2* and *DZNRM2*.)

The calculation of the Euclidean norm is a common numerical computation that is required as a component of many numerical algorithms. A sign of its importance is its inclusion as an elementary generic function, *norm2*, in the Fortran 2008 standard [ISO/IEC 2011]. As we show in Sections 8.2 and 8.3, both the original reference implementation of the Level 1 BLAS routines **NRM2* and their replacements that accompany the latest release of LAPACK (version 3.7.0, December 2016) lose accuracy as the length of the input vector, n, increases. Users expect that the arithmetic operations and basic mathematical functions provided by, say, a compiler or an accompanying math library will deliver results that are close to machine precision. Many would expect the same from low level components like those found in the Level 1 BLAS. With the ability to solve larger problems it is therefore important that the **NRM2* routines should preserve accuracy in the result returned as the length of the vector, n, increases. We propose a new Fortran reference implementation using compensated summation [Higham 2002] that provides almost full machine precision for all n-vectors with $n < N_{max}$ where N_{max} depends on the floating point precision of the arithmetic used to compute the norm.

We have produced two versions of the replacement functions in standalone form and packaged as a module. The standalone code is suitable for replacing the **NRM2* routines in widely available BLAS libraries either at source level or at the linking stage of a compilation. It also allows us to produce a standard conforming interface to these routines from a compliant C compiler via the newly introduced *ISO_C_BINDING* intrinsic module. The module form provides an easy way of providing a generic name, *NRM2*, to the four functions as well as removing the need to type the particular names explicitly in any calling subprograms. For details of all of the intrinsic modules mentioned above see [Metcalf et al. 2011, Chapters 11 and 12].

In Section 2 we look at how differences in floating point hardware and new emerging standards have influenced the design of software. We discuss the requirements of Fortran reference implementations in general along with our design goal for the proposed new **NRM2* routines in Section 3. In Section 4 we provide a detailed overview of the reference codes for computing the l_2 norm of a vector that have appeared over the years and introduce Kahan's algorithm which uses compensated summation to provide improved accuracy in the computed result. We also report on a new implementation of an accurate method for Euclidean norm computation which uses a similar mechanism but is not written as a reference implementation.

Error bounds on the sum of squares with and without compensated summation are discussed in Section 5 along with a comparison of the computational cost of the various methods in terms of floating point operations. New testing and benchmarking suites for Euclidean norm software are introduced in Section 6 and the performance of existing reference implementations is reported in Section 7. We describe our proposed reference implementation in detail and provide performance results in Section 8. Section 9 looks at how existing implementations may be improved by using compensated summation. Differences in the Level 1 BLAS interface and the new Fortran intrinsic function are discussed in Section 10 along with the results we obtained by applying our tests to a current compiler implementation. Finally, Section 11 presents our conclusions.

2. HISTORICAL PERSPECTIVES

Prior to the introduction of hardware that implemented the IEEE arithmetic standard, the results obtained from software using floating point arithmetic were very much dependent upon the platform on which it was being executed. For some floating point implementations single precision arithmetic was so inaccurate that it could not be used for any practical applications and the software needed to be written in double precision. On other platforms the single precision arithmetic was accurate while the double precision implementation was extremely slow and had to be avoided for efficiency reasons; hence the reason for both single and double precision versions of many popular packages (for example, BLAS [Lawson et al. 1979a; Dongarra et al. 1988; Dongarra et al. 1990], LINPACK [Dongarra et al. 1979] and LAPACK [Anderson et al. 1999]).

Portability was also a serious problem. Floating point implementations provided a plethora of different representations and arithmetic parameters (for example, base, mantissa length, exponent range, etc.) as well as different rounding rules. In addition, conditions like underflow and overflow not only occurred at different numerical values but their treatment was not defined by the language standard and was, therefore, left to the whim of hardware designers and compiler writers. Thus underflows were likely to be abruptly set to zero without any feedback to the user while the outcome of a division by zero or an overflow could result in either termination of the executable or a continuation of the computation with, possibly, undefined consequences.

Programmers went to great pains to avoid situations that might result in a computation either terminating due to a system signal or suffering a catastrophic loss of accuracy. In the case of the l_2 norm of a vector, this gave rise to a number of complicated algorithms designed to avoid harmful underflows and unnecessary overflows (see, for example, [Blue 1978] and [Lawson et al. 1979a]).

The arrival of the IEEE floating point standard and its widespread implementation in hardware has alleviated many of these problems. However, the quest for computational speed has had the side effect of sacrificing accuracy. We claim that this loss of accuracy is unnecessary.

3. REFERENCE IMPLEMENTATIONS AND DESIGN GOAL

A reference code may be thought of as a basic level implementation which is designed to allow a software package to be installed easily and for the resulting executable code to solve practical computational problems without an excessive overhead. For problems where optimal efficiency is an absolute requirement it may be necessary to use equivalent platform dependent, hand-crafted, assembler; or, if lower accuracy is acceptable, to use a more efficient but less accurate implementation. However reference codes should always aim to deliver a robust version that will return as accurate a result as possible while taking due regard of efficiency and conformance to the Fortran standard.

Fortran reference implementations may be expected to evolve as the Fortran standard evolves. Indeed we have already seen examples of this happening with the original Level 1 BLAS library where, because of the inclusion of the *incx* argument it was not possible, within the Fortran 66 standard, to dimension some array arguments explicitly. For example, with **NRM2*, the array argument, *x*, should have been declared to be of length *(1+(n-1)*incx)* but integer expressions were not allowed by the prevailing standard. The decision taken was to dimension all arrays to be of length one, a device that was acceptable to most compilers at the time provided array bound checking was disabled. With the arrival of the Fortran 77 standard and the introduction of assumed size arrays it became possible to update the reference library routines to be standard conforming and to allow array bound checking.

The use of standard Fortran as the implementation language also provides a high degree of futureproofing since new standards only very rarely remove features from previous standards and, even when this happens, compilers tend to continue to support them.

Some specific requirements of a Fortran reference implementation in the context of an existing basic library routine are

(1) it can easily replace an existing routine within the library; i.e., it must have exactly the same external interface,
(2) it is portable to any platform that need only support a standard conforming Fortran compiler,
(3) its code should be as efficient as possible and the resultant computation should be as accurate as possible without compromising the overall performance of the package using the routine,
(4) its code should make use of features introduced in newer standards to increase both the robustness and the readability of the routine.

Our *goal* for the proposed replacement Fortran reference routines is

> *to provide standard conforming and portable Fortran routines, *NRM2, for the accurate evaluation of the l_2 norm of a given $n-$vector x. Any non-finite or not-a-number values in the input vector should be detected and the function should return a mathematically correct result (see Section 4.5 for more details of return values for non-finite input).*

This goal is an extension of the original designs as the Level 1 [Lawson et al. 1979a], Level 2 [Dongarra et al. 1988] and Level 3 [Dongarra et al. 1990] BLAS packages are all silent about the handling of floating point exceptions. We advocate using the new IEEE Fortran intrinsic modules to add new features to the **NRM2* routines so as to compute a correct result within the context of IEEE floating point arithmetic and to set IEEE exception flags where appropriate.

4. EXISTING ALGORITHMS AND IMPLEMENTATIONS

In this section we look in some detail at a number of algorithms and reference implementations that are available for computing the Euclidean norm of a vector. We will also report on how these reference codes performed on both a comprehensive test suite and a simple benchmark (for details, see [Hanson and Hopkins 2015]). We concentrate our efforts on Fortran reference codes as we wish to be able to advise on the best algorithm for computing the l_2 norm from both an accuracy and robustness point of view. While we also comment on the performance of the new standard Fortran intrinsic function, *norm2*, available in a number of compilers, these implementations, like vendor supplied versions of the BLAS, must be treated as black-box code, i.e., without access to the source code we have no way of knowing for sure what algorithm has been used or how robust the implementation may be; we can only go on the results of testing. We use execution times for the reference codes to compare the relative efficiency of the various implementations. All timings were obtained from executables generated using the highest level of code optimization available for the compiler we used. While this should produce efficient executable code it should be borne in mind that intimate knowledge of an algorithm can often allow even larger efficiency gains to be made with handcrafted assembler. Thus, whilst timings for the reference codes may provides some insight into the relative efficiency of the underlying algorithms, they should not be taken as an absolute comparison.

To prevent underflows and overflows interfering with the return of a valid, finite result, it is necessary to incorporate some form of scaling when either very small or very large elements appear in the input vector. Thus rather than directly forming

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

we compute

$$S = \sum_{i=1}^{n}\left(\frac{x_i}{scale}\right)^2$$

and form the final result $||x||_2 = scale \times \sqrt{S}$ when all the elements have been processed. The circumstances under which scaling is applied and the value of the scale factor employed differ among implementations and are usually dependent on the floating point arithmetic being used. We consider the following methods most of which are implemented by existing or modified reference codes:

4.1. The Original Algorithm 539 **NRM2* Routines

The original Level 1 BLAS Fortran **NRM2* routines formed part of the software component of Algorithm 539 [Lawson et al. 1979a]. It was hoped that the publication of these codes would lead to more efficient implementations being produced for a wide variety of floating point processors. Indeed assembler versions for the Univac 1108, IBM 360/370 and CDC 6600 were included with the reference code for Algorithm 539 and a version for the 8087 processor was published as Algorithm 653 [Hanson and Krogh 1987].

A more readable version of the Algorithm 539 **NRM2* reference implementation appears in [Hopkins 1996]. This algorithm is one pass and uses a single accumulator to collect the sum of squares of the elements. Scaling only takes place if the current element could potentially cause a catastrophic underflow or overflow in the partial sum. To this end the positive floating point number range is split into three subranges, $[minreal, low]$, $(low, high)$ and $[high, maxreal]$, where definitions and numerical values for low and $high$ are given in Table I for single and double precision IEEE floating point arithmetic. In the original software, values of low and $high$ were used that satisfied a range of floating point number formats in use at the time. In the code used for the testing and benchmarking results given here we replaced the originally published parameter values with ones derived directly from the IEEE standard floating point format definitions.

Table I. Values of the $high$ and low used in the Kahan, Blue and original BLAS routines for IEEE single and double precision formats. In the following ϵ = EPSILON(kind), τ^2 = TINY(kind), $\omega^2 = 2^r$ where r = MAXEXPONENT(kind) $+1$.

	$high$				low		
		sp	dp	γ		sp	dp
Kahan	ω	2^{64}	2^{512}	$\omega/\sqrt{n}$	τ/ϵ	2^{-40}	2^{-459}
Blue	$\omega/\sqrt{\epsilon}$	2^{52}	2^{486}	$\omega/\sqrt{\epsilon}$	τ	2^{-63}	2^{-511}
Alg 539	ω	2^{64}	2^{512}	ω/n	$\tau/\sqrt{\epsilon}$	$2^{-51.5}$	2^{-485}

Scaling occurs if the element of maximum magnitude being processed at any stage of the calculation is greater than zero and either less than low or greater than $\gamma = high/n$. It would appear that the division by n is far too conservative just to prevent an overflow, in a similar situation Kahan [Kahan 1997] uses $\gamma = high/\sqrt{n}$ and, with this

Table II. Values of the scaling factors used in the Kahan, Blue and original BLAS routines for IEEE single and double precision formats. In the following ϵ = EPSILON(kind), τ^2 = TINY(kind), $\omega^2 = 2^r$ where r = MAX-EXPONENR(kind) $+1$.

	$scale_{high}$			$scale_{low}$		
		sp	dp		sp	dp
Kahan	$\sqrt{\epsilon}/\omega$	2^{-76}	2^{-538}	$1/(\tau\epsilon)$	2^{86}	2^{563}
Blue	$\sqrt{\epsilon}/\omega$	2^{-76}	2^{-538}	$1/\tau$	2^{63}	2^{511}
Alg 539	$\max_{1\le i\le j}\lvert x_i\rvert$	—	—	—	—	—

change, the original software performs just as well with our test and benchmark codes. No scaling takes place if all the elements of the input vector have their magnitudes in the middle range; this means that for many practical applications this algorithm is equivalent computationally to the simple loop given in Algorithm 1. These originally published routines use the “Assigned GOTO” statement that was withdrawn as of the Fortran 95 standard although we would expect most compilers to continue to allow this construct for the foreseeable future.

ALGORITHM 1: Simple loop: no checks

```
Input: Vector $x$, size $n$, storage increment $incx$.
Output: The $l_2$ norm of $x$, $||x|| = dnrm2$, returned via the function name.
$sum = 0$;
for $i = 1,\ (n-1)*incx+1,\ incx$ do
    $sum \mathrel{+}= x_i^2$;
end
return ($\sqrt{sum}$);
```

We present pseudocode for this method as Algorithm 2. This, hopefully, provides a more understandable version of the algorithm than both the original software and the restructured source code presented in [Hopkins 1996].

4.2. LAPACK Reference Implementation

This method is credited to Hammarling [Higham 2002, p.500] and is included in the reference version of the Level 1 BLAS available from netlib [Dongarra and Grosse 1987]; it is also provided in the BLAS directory that has formed part of multiple releases of the LAPACK library [Anderson et al. 1999], including the current release (version 3.7.0, December 2016). The method used may be regarded as a simplified and more portable version of the Algorithm 539 methods described in Section 4.1. Improved portability is obtained by always setting the scale factor to be $\max\{|x_j|\}_{j=1}^{i}$ when processing element i of the input vector, thus removing the need to predefine any subranges based on the underlying machine arithmetic or the type of the input vector.

Pseudocode for this algorithm is given in Algorithm 3. As with the Algorithm 539 method this is a one pass method that uses a single accumulator. We note that the scaling factor is changed whenever the current element is larger in magnitude than all its preceding elements. This means that, in the extreme case, when the input vector consists entirely of elements whose absolute values are increasing, the partial sum will be rescaled as each element is processed.

ALGORITHM 2: Readable version of the original BLAS Level-1 routine from Algorithm 539

```
**Input**: Vector $x$, size $n$, storage increment $incx$.
**Output**: The $l_2$ norm of $x$, $||x|| = dnrm2$, returned via the function name.
!
! The definitions and values for $low$ and $\gamma$ are given in Table I
!
$i = 1$; $sum = 0$; $nn = 1 + (n-1) * incx$;
! Ignore leading zero elements
while $x_i == 0$ do
    $i$ += $incx$; if $i > nn$ then return (0)
end
! $sc$ is the current value of the scaling factor. It is updated in $ScaledUpdate$
$sc = 0$;
! Lower range: scale while $|x_i| \leq low$
while $|x_i| \leq low$ do
    $ScaledUpdate$; $i$ += $incx$;
    if $i > nn$ then return ($sc * \sqrt{sum}$)
end
! Mid range: scale current sum; no scaling performed while
! elements remain in this range
$sum = (sum * sc) * sc$;
while $|x_i| < \gamma$ do
    $sum$ += $x_i^2$; $i$ += $incx$;
    if $i > nn$ then return ($\sqrt{sum}$)
end
! High range: scale current sum; scale all remaining elements
$sc = |x_i|$; $sum = 1 + (sum/sc)/sc$; $i$ += $incx$;
while $i \leq nn$ do
    $ScaledUpdate$; $i$ += $incx$;
end
return ($sc * \sqrt{sum}$);

procedure $ScaledUpdate$;
Input: Current element, $x_i$, scale factor, $sc$, and partial sum, $sum$.
Output: Updated partial sum and, possibly, updated scale factor, $sc$.
if ($|x_i| > sc$) then
    $sum = 1 + sum * (sc/x_i)^2$; $sc = |x_i|$;
else
    $sum$ += $(|x_i|/sc)^2$;
end
```

Higham [Higham 2002, p.571] shows that the computation of the return value, $(sc * \sqrt{sum})$, can only overflow if $||x||_2$ exceeds the largest storable floating point number available in the precision being used.

4.3. Blue's Algorithm

Like the Algorithm 539 method, Blue's Algorithm [Blue 1978] splits the positive floating point range into three subranges and uses scaling for elements in the lower and upper intervals. Its main difference is its use of three accumulators to gather a separate partial sum for the subset of elements falling within each subrange. The other major difference with the two preceding algorithms is that the scaling factors used are dependent on the parameters of the floating point arithmetic and do not vary accord-

ALGORITHM 3: Lapack version

```
Input: Vector $x$, size $n$, storage increment $incx$.
Output: The $l_2$ norm of $x$, $||x|| = dnrm2$, returned via the function name.
$sum = sc = c = 0$;
for $i = 1,\ (n-1)*incx+1,\ incx$ do
    if ($|x_i| == 0$) then cycle;
    $ScaledUpdate$; ! see Algorithm 2 for details
end
return ($sc * \sqrt{sum}$);
```

ing to the contents of the input vector. The values used for low and $high$ along with the respective scaling factors are given in Tables I and II. A pseudocode version of the original algorithm may be found in [Blue 1978, page 16].

The original algorithm collects all three accumulators over all the data elements. However, if element values in the high range are present then the low range accumulator does not contribute to the final computed result. The only circumstances under which the low range accumulator can contribute is if there are no values in the high range. Thus only one scaled accumulator (for either the low or high range) is required; the pseudocode in Algorithm 4 illustrates how this variant may be implemented. For certain data vectors this approach saves performing arithmetic operations that do not contribute to the final result.

Since we could find no freely available Fortran implementation of Blue's algorithm, we have implemented Fortran reference module containing all four **NRM2* routines for both variants to allow for accuracy and timing comparisons.

4.4. The *FaithfulNorm* Routine

Recently a new algorithm, *FaithfulNorm*, was proposed [Graillat et al. 2015] for computing the l_2 norm of a vector with guaranteed accuracy to within a single bit of the floating point type being used (double precision in their implementation). This algorithm treats the data in a very similar way to the Blue and Kahan methods; three ranges are defined and elements whose absolute values fall outside the middle range are scaled prior to the accumulation of their squares. They use a two accumulator method similar to the one employed in Algorithm 4.

High accuracy is obtained in the accumulation process by using a pair of floating point variables for each of the accumulators thereby effectively doubling the precision available to compute the sums of squares. The implementation of this part of the algorithm uses a variation of the double-double arithmetic used in [Hida et al. 2007] that reduces the number of basic floating point operations required to compute the Euclidean norm to $11(n-1)$. The error bound for their method is $3n\epsilon^2/(1-3n\epsilon^2)$ which is comparable to Kahan's method using compensated summation (see Section 5 for more details).

The software package provided by the authors [Graillat et al. 2015] contains a reference implementation of the double precision function (a single precision version is discussed in the article but no implementation is provided) which is written in C and requires the GNU MPFR Library [MPFR 2016] to be installed. Additionally two SIMD versions are provided for Intel and AMD processors using the x86 instruction set with and without Advanced Vector Extension (AVX) support.

While the new C interoperability features available from Fortran 2003 would allow a Fortran wrapper routine to be constructed and it would be straightforward to create the four required **NRM2* functions, neither of these versions of *FaithfulNorm* fits the

ALGORITHM 4: Blue's Algorithm using Only Two Accumulators

```
Input: Vector $x$, size $n$, storage increment $incx$.
Output: The $l_2$ norm of $x$, $||x|| = dnrm2$, returned via the function name.
!
! The definitions and values for $low$ and $\gamma$ are given in Table I.
! $R$ is the largest representable floating point value.
!
$i = 1$; $sum = a_{med} = a_{other} = 0$; $nn = 1 + (n - 1) * incx$;
$overFlowLimit = R * scale_{high}$;
! $bigElt$ is used to flag the presence of elements in the high range
$bigElt = false$;
! Loop through elements. Set $bigElt$ to $true$ when first high range element found
while $i <= nn$ do
    if $|x_i| > \gamma$ then
        if $bigElt$ then ! High range values already seen
            $a_{other} += (x_i * scale_{high})^2$;
        else
            ! First large element; reinitialise scaling accumulator
            $bigElt = true$; $a_{other} = (x_i * scale_{high})^2$;
        end
    else if $|x_i| < low$ then ! Low range values
        if $\neg bigElt$ then $a_{other} += (x_i * scale_{low})^2$;
        ! Low range values can't contribute
    else ! Middle range element
        $a_{med} += x_i^2$
    end
    $i += incx$
end
if $a_{other} == 0$ then return $(\sqrt{a_{med}})$; ! Only middle range values
! Scaling required -- get scale factor
if $bigElt$ then
    $sc = scale_{high}$
else
    $sc = scale_{low}$
end
if $\sqrt{a_{other}} > overFlowLimit$ then return $(\infty)$; ! Result not representable
! We must now have a representable result
if $a_{med} == 0$ then return $(\sqrt{a_{other}}/sc)$; ! Only scaled accumulator is non-zero
! $a_{med}$ may contribute
$y_{min} = \min(\sqrt{a_{med}}, \sqrt{a_{other}}/sc)$;
$res = \max(\sqrt{a_{med}}, \sqrt{a_{other}}/sc)$;
if $y_{min} >= \sqrt{\epsilon} * res$ then
    return $(res * \sqrt{1 + (y_{min}/res)^2})$;
else
    return $(res)$;
end
```

requirements of a Fortran reference implementation (discussed in Section 3) suitable for replacing existing **NRM2* routines within the Level 1 BLAS Library because

(1) a compatible C compiler is required to compile the available assembler implementation,
(2) an additional library, *MPFR*, is required if no appropriate assembler version is available. This in turn would require the availability of a compatible C compiler,

(3) use of assembler to gain execution speed means that the resultant implementation is neither portable nor futureproof.

because we are interested only in Fortran reference routines in this article, we do not consider this implementation any further.

4.5. Kahan's Algorithm

This method has been suggested by Kahan [Kahan 2012] and was originally written as a Matlab code. A pseudocode version of this method is presented in Algorithm 5. Unlike the other methods described in this section it is designed to deal with non-finite values in the input vector. This leads to the input vector being scanned twice if all the input elements are finite floating point numbers and scaling is required to form the sum of squares safely. As with methods discussed in Sections 4.1 and 4.3 the floating point range is divided into three subranges with the values used for low and $high$ given in Table I for single and double precision IEEE floating point. The scaling factors used are given in Table II.

By increasing the number of logical tests required to process each element of the vector we can avoid a second pass through the data. The pseudocode for this version is given in Algorithm 6.

The major difference over all the methods described above is the use of *compensated summation* (see [Higham 2002] for details) to accumulate the sum of squares; this has the important quality of ensuring that the computed sum is obtained very accurately without the explicit need for extended precision arithmetic (see Section 5 for more details).

We also need to consider what result should be returned if the input vector contains non-finite values; the IEEE 754 standard defines formats for IEEE floating point representations of infinite values of either sign and NaNs—floating point values not specified as the result of an operation. Kahan comments [Kahan 1997] that NaN has historical precursors in Konrad Zuse's "Undefined", and Seymour Cray's "Indefinite". These were intended to allow both the programmer and the hardware to defer judgment about an exception when only some of a vector's values would be used following a vectorized computation.

What follows is based on personal emails with W. Kahan. If the input vector contains

(1) one or more $\pm Inf$ values but no NaNs, then the result is $+Inf$,
(2) one or more NaN values (signalling or quiet) but no Inf values, then the result is $qNaN$,
(3) one or more $\pm Inf$ values and one or more NaN values, then the result is $+Inf$.

This is based on the following argument. If an element of an array is infinite, then its l_2 norm is also infinite, regardless of whether one or more elements are NaNs. This must be so because, if the NaN were replaced by any non-NaN value, the norm would be infinite. In other words, if a function $f(s,t)$ takes a value $f(Inf,t)$ that is independent of t, finite or not, then $f(Inf, NaN)$ must be that value too. We apply this to the sum of squares of the components of a vector x. Partition the elements of x into two sets: S_1 containing all the non-NaN values including both finite and infinity values, and S_2 containing all the NaN values present.

Let s and t be the sum of squares of the elements of S_1 and S_2 respectively. If S_1 contains at least one infinity value then $s = Inf$. If S_2 is non-empty then $t = NaN$ and the comments above imply that the l_2 norm is also infinite; otherwise S_2 is empty, $t = 0$ and $s + t$ is infinity. In both cases, the result is infinity.

ALGORITHM 5: Kahan's Original IEEE Algorithm

```
Input: Vector $x$, size $n$, storage increment $incx$.
Output: The $l_2$ norm of $x$, $||x|| = dnrm2$, returned via the function name.
!
! The definitions and values for the floating point parameters $low$, $\gamma$, $scale_{low}$
! and $scale_{high}$ used here are given in Tables I and II
!
$sum = ahat = s = 0$; $nanInInput = FALSE$;
for $i = 1,\ (n-1)*incx+1,\ incx$ do
    if ($x_i$ does not test as finite) then
        if ($x_i$ is a $NaN$) then
            $nanInInput = TRUE$; $s = NaN$; cycle ; ! on i loop
        else
            ! Know result is Infinity -- set result and exit immediately
            return $(+\infty)$;
        end
    end
    if ($nanInInput$) then cycle; ! Only looking for $Inf$ values
    ! $NaN$ will propagate here providing it is not a signalling $NaN$
    $ahat = \max\{ahat, |x_i|\}$;
    if $ahat \geq \gamma$ then cycle; ! on i loop
    ! Perform compensated summation
    $s \mathrel{+}= x_i^2$; $t = sum$;
    $sum = t + s$; $s = (t - sum) + s$;
end
if ($nanInInput$) then return $(NaN)$;
if $ahat \geq \gamma$ then
    $sc = scale_{high}$;
else if $ahat \leq low$ then
    $sc = scale_{low}$;
else
    ! Normal case - no scaling
    return $(\sqrt{sum})$;
end
$sum = s = 0$;
! Scaling required, rescan the input vector
for $i = 1,\ (n-1)*incx+1,\ incx$ do
    $s \mathrel{+}= (x_i * sc)^2$; $t = sum$;
    $sum = t + s$; $s = (t - sum) + s$;
end
! The following statement might overflow
return $(\sqrt{sum}/sc)$;
```

The Fortran 2008 standard [ISO/IEC 2011, Section 13.7.1] states that when the *IEEE_ARITHMETIC* intrinsic module is available for use with standard intrinsic procedures

> If an infinite result is returned, the *IEEE_OVERFLOW* or *IEEE_DIVIDE_BY_ZERO* shall signal; if a NaN result is returned, the flag *IEEE_INVALID* shall signal. Otherwise, these flags shall have the same status as when the intrinsic procedure was invoked.

Table III shows the various possible input vector contents along with the results returned and changes made to the IEEE floating point status flags by the proposed new

ALGORITHM 6: One Pass Version of Kahan's IEEE Algorithm

```
Input: Vector $x$, size $n$, storage increment $incx$.
Output: The $l_2$ norm of $x$, $||x|| = dnrm2$, returned via the function name.
! The definitions and values for the floating point parameters $low$, $\gamma$, $scale_{low}$
! and $scale_{high}$ used here are given in Tables I and II
$sum = s = 0$; $nanInInput = FALSE$; $range = lowR$; $sc = scale_{low}$;
! Set meaningful names for possible ranges.
$lowR = 0$; $mediumR = 1$; $highR = 2$;
for $i = 1,\ (n-1)*incx+1,\ incx$ do
    if ($x_i$ does not test as finite) then
        if ($x_i$ is a $NaN$) then
            $nanInInput = TRUE$; $s = NaN$; cycle ; ! on i loop
        else
            ! Know result is Infinity -- set result and exit immediately
            return $(+\infty)$;
        end
    else
        ! Process finite values
        if ($nanInInput$) then cycle; ! Only looking for $Inf$ values
        ! Summing finite values
        if ($range == lowR$) then
            if ($|x_i| \leq low$) then ! All values so far in low range
                $s = s + (x_i * sc)^2$;
            else if ($|x_i| \geq big$) then ! Move to high range processing
                ! Any low range values collected so far may be discarded
                $sc = scale_{high}$; $sum = 0$; $s = (x_i * sc)^2$; $range = highR$;
            else ! Move to medium range processing
                $s = s/sc^2 + x_i^2$; $sum = sum/sc^2$; $sc = 1$; $range = mediumR$;
            end
        else if ($range == mediumR$) then
            if ($|x_i| < \gamma$) then ! Stay in medium range
                $s = s + x_i^2$;
            else ! Move to high range processing
                $sc = scale_{high}$; $s = s * sc^2 + (x_i * sc)^2$;
                $sum = sum * sc^2$; $range = highR$;
            end
        else ! Processing high range
            $s = s + (x_i * sc)^2$;
        end
        ! Perform compensated summation
        $t = sum$; $sum = t + s$; $s = (t - sum) + s$;
    end
end
if ($nanInInput$) then return $(NaN)$;
! The following statement might overflow
return $(\sqrt{sum}/sc)$;
```

reference procedure. We note here that *IEEE_OVERFLOW* is only set if the resultant l_2 norm exceeds the maximum representable value; in this case the inexact flag also signals. If the input vector contains any $\pm Inf$ values then the result is taken to be exactly $+Inf$ and no overflow flag is set. We believe that this behaviour reflects the mathematical qualities of the underlying function.

Table III. Return values and IEEE floating point status flag settings for various input vector data to the proposed new reference procedures

Input Vector Contains	Result	Status Flags Set
one or more NaN; no Inf values	$qNaN$	invalid
one or more NaN; one or more Inf values	$+Inf$	none
no NaN; one or more Inf values	$+Inf$	none
All finite floating point numbers		
a) Result $>$ HUGE($\cdot$)	$+Inf$	inexact, overflow
b) Result $\leq$ HUGE($\cdot$)	finite fp value	inexact may be set

5. ERROR BOUNDS AND OPERATION COUNTS

In the worst case scenario use of any of the algorithms described in Sections 4.1 to 4.3 may lead to the last $\log_2(n)$ bits of the binary result (or the last $\log_{10}(n)$ digits of its decimal equivalent) being corrupted [Graillat et al. 2015]. This means that all of these algorithms gradually lose accuracy as n, the number of elements in the input vector, increases.

Higham [Higham 2002, p.85] gives the forward error bound for computing $S_n = \sum_{i=1}^{n} y_i$ using compensated summation as

$$E_n = |S_n - \hat{S}_n| \leq (2u + (nu^2)) \sum_{i=1}^{n} |y_i|$$

where $\hat{S}_n$ is obtained using floating point arithmetic and u is the unit round-off error defined to be $\epsilon/2$ where ϵ, used in Tables I and II, is the machine epsilon (the distance between one and the next largest representable floating point number).

For the Euclidean norm computation for real vectors we have $S_n = \sum_{i=1}^{n} x_i^2$ so $\sum_{i=1}^{n}|x_i^2| = |\sum_{i=1}^{n} x_i^2| = S_n$ which guarantees to yield a small relative error bound since

$$\frac{E_n}{S_n} \leq 2u + O(nu^2)$$

and ([Higham 2002, p.85]) provided $nu < 1$ the bound is independent of n. For IEEE arithmetic this requires $n < 1.7 \times 10^7$ (single precision) and $n < 9 \times 10^{15}$ (double precision). For complex vectors of length n, we can apply the same analysis to the real vector of length $2n$ where x_{2i-1} is the real part and x_{2i} is the imaginary part of the i^{th} complex element.

Table IV provides a summary of the operation counts required for each of the Fortran reference implementations described in Section 4. To allow a fair comparison with the Kahan and Hybrid methods we have assumed that only finite floating point values appear in the input vector and we have ignored any operations required to detect non-finite values. The 'best' case generally occurs when all the element values lie in a range that requires no scaling; the exception is the LAPACK algorithm which requires the element of largest magnitude as the first element. The worst case occurs when all the elements require scaling to take place; this causes extra multiplies and, in the original BLAS and LAPACK methods, extra divides. For Blue's algorithm a slightly more expensive case occurs where $n-1$ element values all lie either in the low or high ranges and the remaining value is in the middle range and is large enough to force both accumulators to contribute to the final computed result.

A worst case example for Kahan's original definition (Algorithm 5) occurs when all the elements are in the low and medium ranges apart from the last which is in the high range. The algorithm then effectively collects the sum of squares with compen-

Table IV. Operation Counts for Fortran Reference Routines

	Best Case					Worst Case				
Algorithm	$+$ or $-$	$\times$	$/$	$\surd$	*SCALE*	$+$ or $-$	$\times$	$/$	$\surd$	*SCALE*
Simple Loop	n	n	$-$	1	$-$	n	n	$-$	1	$-$
Original BLAS	n	n	$-$	1	$-$	n	$2n+1$	n	1	$-$
LAPACK	n	$n+1$	n	1	$-$	n	$2n+1$	n	1	$-$
Blue's Algorithm	n	n	$-$	1	$-$	$n+1$	$n+2$	1	3	$n+1$
Kahan Original	$4n$	n	$-$	1	$-$	$8n-4$	$2n-1$	$-$	1	n
Kahan One Pass	$4n$	n	$-$	1	$-$	$5n$	n	$-$	1	$n+3$
Hybrid + One Pass Kahan	$4n$	n	$-$	1	$-$	$9n$	$2n$	$-$	1	$n+3$

sated summation twice; summing the first $n-1$ elements during the first pass and then having to make a complete second pass to perform a scaled summation. With the one pass variation we use the partial sum of the first $n-1$ elements and rescale to accommodate the final value. We, therefore, recommend the use of the one pass version for the hybrid method.

Also, because the scaling factors are chosen to be integral powers of two, floating point divides may be replaced in both the Blue and Kahan algorithm by calls to the new Fortran intrinsic function, *SCALE*, which just adjusts the exponent and should be more efficient than a full divide operation.

We see from Table IV that the use of compensated summation by the Kahan and hybrid methods increases the number of floating point add/subtract operations required from n to $4n$ when applied to simple data requiring no scaling. If scaling is required by the hybrid method, then the number of floating point adds/subtracts and multiplies rises to $9n$ and $2n$ respectively although we are able to trade calls to the *SCALE* intrinsic against n divisions when comparing to the Original BLAS and LAPACK implementations.

6. TESTING

We generated a comprehensive test suite of around fifty examples which were designed to test implementations of the **NRM2* routines as exhaustively as possible. We believe that passing all these tests should instil a good level of confidence in the code under test. Full details of all the tests that comprise the test suite may be found in [Hanson and Hopkins 2015].

To check the accuracy of the computed results we use an oracle routine to compute the norm via a simple square and add loop using a higher precision than that used to store the input data. For single precision input we implement the oracle routine in double precision where, for IEEE arithmetic, the increased mantissa length and exponent range ensure that the final result can always be computed accurately and without destructive underflow or overflow occurring. For double precision data the 'obvious' way of implementing the oracle would be to use quadruple precision. However, this has the drawback that quadruple precision is not part of the Fortran standard so either it may not be available at all or it will be implemented in a compiler dependent way. This means that, for example, we have no guarantee that the compiler dependent exponent range will have been increased enough to prevent overflows using a simple square and add loop; the NAG Fortran compiler (v6.1, build 6106) is a case in point, the mantissa length is increased to 106 bits but the exponent range, $(-968, 1023)$, is less than for double precision. We thus use the multiple precision package [Bailey 1995] where the exponent range is $(-maxint, maxint)$ and we set the precision to be twice that of IEEE double precision.

7. PERFORMANCE OF EXISTING IMPLEMENTATIONS

One of the problems with using a single accumulator for the sum of squares is that, without some means of extending the precision of the calculations, some forms of input data will always cause problems. For example, consider the following input vector (Test 36 in our test suite [Hanson and Hopkins 2015])

$$\{x\}_{i=1}^{n} = \{1, p, p, \cdots, p\}$$

where the value of p is chosen so that, for the floating point arithmetic being used, $1 + p^2$ returns one. The vector is then made large enough so that $1 + (n-1)p^2$ would be representable.

Using the original Level 1 BLAS and LAPACK reference implementations the result is returned as one whereas Blue's algorithm returns a more accurate result since the $\sum p^2$ is collected in a separate accumulator. Kahan's method also returns the correct result due to the use of compensated summation.

We note that both the original and LAPACK methods do generate an accurate result if x_1 and x_n are swapped; i.e., $\sum p^2$ is collected before the one is added (Test 37 in the test suite).

Only Blue and Kahan's algorithms attempt to detect a true overflow condition, i.e., a test for overflow of the result forms part of the algorithm. Our implementation of Blue's Algorithm takes advantage of this by explicitly setting the returned result to $+Inf$ and the inexact and overflow flags to signal. In all other circumstances all the methods discussed in Sections 4.1 to 4.3 are allowed to return the result and status flag settings without any explicit tests or extra code being added. The vast majority of the problems detected by our test package are caused by status flags not signalling as expected (for example, the invalid flag does not signal when a NaN is returned and the value returned for an input vector containing at least one Inf and one NaN value is NaN rather than $+Inf$).

One reason for the benchmark test is to ascertain how accurate the computed results are for vectors of increasing length. We first consider 'easy' problems where the input vectors of length 10^n for $n = 1, 2, \ldots, 5$ are used with all the elements assigned random values in the range $(0, 1)$. The maximum and average relative errors, presented in Table V, are given in *ulps* (multiples of the unit round off error) $\epsilon/2$, to emphasize the loss of accuracy. We are interested in how rounding errors may affect the computed results as the number of elements in the input vector increases.

The original Level 1 BLAS, the LAPACK and Blue's algorithms all produced average and maximum relative errors that increased by a factor of approximately three for each increase in n by a factor of 10. As expected the results from the original Level 1 BLAS and Blue's algorithms are very similar since, for these input vectors, they are computing the results via a simple square and add loop.

For the lengths of vectors used in the benchmarking runs using both single and double precision arithmetic Kahan's method returned results with a maximum relative error of one ulp when compared to the oracle values. This clearly illustrates the efficacy of using some form of extended precision computation when collecting the sum of squares.

In Section 9 we consider how the accuracy of existing implementations may be improved by the use of compensated summation.

8. THE NEW HYBRID ROUTINES

We use the new features available via Fortran's new IEEE arithmetic modules to detect and report both the presence of NaN and Inf values in the input vector as well as genuine overflow conditions where the input vector contains valid, finite values but the resultant value of the l_2 norm is greater than the maximum storable value.

Table V. Average and maximum relative error given as multiples of the unit round off error for simple random input vectors with all elements in the range $(0, 1)$ using the three versions of the **NRM2* routines provided by the original BLAS [Lawson et al. 1979a], Blue's method [Blue 1978] and the reference BLAS supplied with LAPACK [Project 2015]

$\log_{10} n$	Original Blas				Blue's Algorithm				LAPACK			
	Single		Double		Single		Double		Single		Double	
	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
Real												
1	0.1	2	0.1	3	0.1	3	0.1	3	0.2	3	0.2	3
2	0.6	6	0.6	5	0.6	6	0.6	5	0.6	6	0.6	7
3	2.8	17	2.8	16	2.8	17	2.8	16	2.8	18	2.8	16
4	9.5	48	9.2	40	9.5	48	9.2	40	9.5	52	9.2	46
5	82.3	172	27.9	109	82.3	172	27.9	109	82.9	176	27.2	151
Complex												
1	0.2	3	0.2	3	0.2	3	0.2	3	0.3	4	0.3	4
2	0.9	9	0.9	8	0.9	9	0.9	8	1.0	9	1.0	8
3	4.1	24	4.1	27	4.1	24	4.1	27	4.2	30	4.2	27
4	14.8	70	13.2	64	14.8	70	13.2	64	14.8	69	13.4	66
5	238.6	368	38.4	188	238.6	368	38.4	188	238.8	380	40.3	215

Kahan's algorithm provides an excellent basis for achieving the goal specified in Section 3. The use of compensated summation ensures a very accurate result when compared to the other methods described in Section 4. Its only downside is its execution time efficiency in the case where all the elements of the input vector lie in the middle range defined by the original Level 1 BLAS and Blue's algorithm. We assert that this is the most commonly occurring input in practice and results in all the methods in Section 4, with the exception of the LAPACK reference implementation, performing no scaling at all on the input vector elements.

We, therefore, propose a hybrid approach to the computation (a general strategy of the type proposed is mentioned in [Metcalf et al. 2011, Chapter 11, p. 237-238]). For the most commonly occurring case we use a simple loop with compensated summation to ensure high accuracy at low cost (for pseudocode, see Algorithm 7). We then check for either a NaN or Inf result; or for the IEEE underflow exception flag signalling. If none of these conditions have occurred we are done; otherwise we recompute using Kahan's method which requires at least a second pass through the data. We choose to implement the one pass version of Kahan's method (Algorithm 6) to reduce the number of additional floating point operations required if the input data needs scaling. A pseudocode version of the proposed hybrid method is given as Algorithm 8.

ALGORITHM 7: Simple loop using compensated summation to preserve accuracy

```
Input: Vector $x$, size $n$, storage increment $incx$.
Output: The $l_2$ norm of $x$, $||x|| = dnrm2$, returned via the function name.
$sum = s = 0$;
for $i = 1,\ (n-1)*incx+1,\ incx$ do
    $s \mathrel{+}= x_i^2$; $t = sum$;
    $sum = t + s$; $s \mathrel{+}= (t - sum)$;
end
return $(\sqrt{sum})$;
```

This approach may avoid scaling which would have occurred with all the methods described in Section 4. Scaling only occurs in the hybrid algorithm if the result of

ALGORITHM 8: IEEE hybrid method

```
Input: Vector $x$, size $n$, storage increment $incx$.
Output: The $l_2$ norm of $x$, $||x|| = dnrm2$, returned via the function name.
Clear UNDERFLOW flag;
Compute $nrm2$ using Algorithm 7;
if the UNDERFLOW status flag is signaling OR the result is either $NaN$ or $Inf$ then
    Clear flags;
    Use Kahan's $l_2$ algorithm to compute $nrm2$;
end
```

the simple loop (Algorithm 7) actually causes an overflow or underflow exception. For example, if an input vector has just one element equal to γ then the two versions of Kahan's method (see Algorithms 5 and 6) will scale whereas the simple loop of the hybrid algorithm will not. We also note that for double precision and a vector of length 10^8 at least one element would need to be $> 10^{150}$ or $< 10^{-138}$ in magnitude to trigger scaling using Kahan's method. For the simple loop with compensated summation used in the proposed hybrid method these values would need to be even more extreme to cause Kahan's method to be called.

Finally, for random data vectors where all the elements are in the range $(0, 1)$ in magnitude, the hybrid method provides the same accuracy on the benchmark tests (see Section 7 above) as Kahan's algorithm since for the input data used Kahan's algorithm reduces to Algorithm 7.

8.1. Reference Software

We preserve the naming convention used in the Level 1 BLAS and detailed in Section 1. These four routines are provided both as separate functions, suitable for replacing an existing reference implementation within a pre-compiled library, and as a Fortran 2008 module, `nrm2HybridMod`, with the functions as module procedures. If the module is used to override or supersede the use of a pre-compiled library then it is necessary to remove all type declarations to `SNRM2`, `DNRM2`, `SCNRM2` and `DZNRM2` from the application code.

Include files have been used to reduce the repetition of source code that is common to more than one routine. This feature is now part of Fortran 2008 and may be used to improve maintainability; but there is a balance to be struck between reducing duplicated code to an absolute minimum and having source code that is easily human readable. Thus we have limited the number of include files used within a single function to two and the depth of include files to one (i.e., include files do not contain include files). Typically include files allow us to share code between the single and double precision versions although different files are required for real and complex input data.

We also took advantage of the C interoperability features in the new Fortran standard to provide a set of interface routines that would allow C software to call the Fortran **NRM2* routines. These are general purpose in that they could be linked to any Fortran implementation of the **NRM2* routines using the specific Fortran names.

The proposed hybrid method produced results which agreed with the expected values of the l_2 norm for all the tests in the test suite. Additionally, during the extensive runs of the benchmark tests the results generated by this implementation were always correct to within one ulp when compared to the oracle values. These results make the case for the use of compensated summation to improve the accuracy of the computed result.

The IEEE status flags returned during the running of the test suite were as predicted except for two of the tests. Tests 5 and 6 ($n = 1$ with $x = [HUGE(\cdot)]$ and

$x = [-HUGE(\cdot)]$ respectively) reported that the *IEEE_INEXACT* was signalling. This occurs because vectors of length one are not treated as special cases and *HUGE(.)* is not a power of two. The squaring of the scaled value (all the mantissa bits are one) causes the inexact flag to signal even though the subsequent square root produces a result which is correct to the last bit. For more discussion of the case $n = 1$ see the user manual [Hanson and Hopkins 2015].

It is vital that compiler optimization honors the parentheses in the compensated summation algorithm and does not alter the intended computation by making an algebraic substitution of $s = 0$ at the end of the loop in Algorithm 7. Such an optimization is not standard conforming [ISO/IEC 2011, Note 7.31]. The Intel compiler XE 14.0.2 needed the option *fp:precise* to avoid this non-standard substitution.

8.2. Benchmark Summary of dnrm2

Table VI provides a timing comparison for easy data for various versions of the competing implementations. The data given is the limiting value of the ratio of the time taken for the competing implementation to the time taken for the proposed hybrid method, as the length of the vector, n, increases. Values greater than one indicate that the hybrid method is more efficient. We also give the maximum relative error (MRE) in ulps in the computed solution. The data used is all in the range $(0, 1)$ as described in Section 7.

These results clearly show the gains in accuracy obtained through using compensated summation (CS); all the basic implementations described in Sections 4.1–4.3 return results that are within one or two ulps when CS is incorporated. The very small errors seen in the LAPACK reference implementation are removed by using a scaling factor that is a power of two, which causes no rounding errors, rather than one of the input data values (see Section 4.2 for details).

For the data sets used, the hybrid method is essentially a square and add loop with compensated summation. The cost of the accuracy improvement through the use of CS is a factor of between 3.5 and 2 for the original Level 1 BLAS and Blue's implementation respectively. As expected, adding CS to these methods results in almost the same calculations taking place and, therefore, almost the same overall timings.

8.3. Effect on Accuracy of Increasing Vector Length

To quantify the effect on accuracy of increasing n we ran the single precision, real routine, *SNRM2*, using both the proposed hybrid method (Algorithm 8) with compensated summation and the LAPACK reference implementation (Algorithm 3) on random data in the range $(0, 1)$ for $n = 10^r$ for $r = 1 \ldots 9$. The oracle result was computed with a simple square and add loop using the quadruple precision available under the NAG compiler (106 bit mantissa, exponent range $(-968, 1023)$). For all the vectors generated the use of compensated summation returned a result that agreed with the oracle to within one ulp even when the vector length exceeded N_{max} for IEEE single precision (see Section 5). Table VII shows how the maximum recorded number of bits lost in the computed value using the current LAPACK reference implementation increased steadily with n; for $n > N_{max}$ (IEEE single precision) no significant decimal digits are returned.

9. UPGRADING EXISTING IMPLEMENTATIONS

The accuracy of all the methods described in Sections 4.1 to 4.3 can be improved by utilizing the compensated summation procedure for collecting the sums of squares. This is straightforward to implement; for the original Level 1 BLAS and LAPACK implementations we have only to rescale the compensation term whenever the partial sum of squares is rescaled. For Blue's algorithm we need to use separate compensation

Table VI. Timing and accuracy results for a variety of **NRM2* implementations on random data in the range $(0, 1)$. Accuracy is recorded as the maximum error in *ulps* recorded. All timings are normalized by dividing by the time taken by the proposed hybrid method on the same data. The values given are for $n = 10^5$.
In the Table *+CS* signifies that compensated summation has been added to the basic Algorithms given in Section 4.

	Real		Complex		MRE
	sp	dp	sp	dp	ulps
Section 4.1					
Original Alg 539	0.29	0.29	0.46	0.46	404
Restructured	0.30	0.30	0.26	0.26	404
Restructured + CS	0.99	1.00	1.00	0.99	1
Section 4.2					
Original Lapack/netlib	0.58	1.16	0.60	1.17	392
+ CS	1.00	1.16	1.00	1.19	3
+ CS + power of 2 scaling	1.00	1.00	1.00	1.00	1
Section 4.3					
Original Blue	0.75	0.75	0.75	0.75	404
+ CS	1.50	1.50	1.50	1.50	1
Section 4.5					
Kahan (one pass)	1.74	1.76	1.99	1.98	1
Fortran intrinsic					
norm2 (Nag)	0.50	1.08	0.93	1.68	1

Table VII. Growth of the relative error in the computed Euclidean norm using the LAPACK reference implementation on random real data in the range $(0, 1)$ for $n = 10^r$. E_m is the maximum relative error detected in the computed value for a given value of n, $\lceil \log_2(E_m) \rceil$ is the number of bits of accuracy lost for this value of E_m and $\lceil \log_2(10^r) \rceil$ is an upper bound for this value (see Section 5). For practical purposes this cannot exceed 24 for IEEE single precision arithmetic.

r	1	2	3	4	5	6	7	8	9
$\lceil \log_2(E_m) \rceil$	1	2	4	5	7	11	16	22	23
$\lceil \log_2(10^r) \rceil$	4	7	10	14	17	20	24	24	24

terms; one for each accumulator. When this is done the accuracy of all the implementations for random input data in the range $(0, 1)$ is improved to the same level as Kahan's algorithm. Both the LAPACK and, when the input data triggers scaling, the Level 1 BLAS reference implementations may still generate slightly larger errors; this may be removed by using a scaling factor that is a power of two rather than the absolute value of the largest element. Versions of all these upgraded routines are included in the accompanying software package.

9.1. Specific detail for changes to Algorithm 539

We have produced a replacement module for the routines that appeared in the original Algorithm 539 [Lawson et al. 1979a]. This implementation is the same computational algorithm but differs from the original in the following ways:

(1) the code has been carefully restructured to make it more readable than both the original and the proposed improvements made in [Hopkins 1996].

(2) all squares of variables are computed using multiplication rather than exponentiation; with some compilers this can give a small improvement in accuracy for larger input vectors.
(3) the original *cutlo* and *cuthi* values have been replaced by values computed specifically for IEEE floating point arithmetic rather than using a 'worst case' setting.
(4) the value of the local variable *hitest*, originally set to *cuthi*/*n* has been replaced by *cuthi*/*SQRT(n)* which increases the length of the middle subrange $(low, high)$ in which no scaling is employed without increasing the risk of an avoidable overflow occurring.

10. NEW FORTRAN STANDARD INTRINSIC FUNCTION, *NORM2*

The 2008 standard introduced a new generic, intrinsic function, *norm2*, to compute the l_2 norm of real arrays. This function takes a single, one-dimensional real array argument and returns a result of the same kind as the vector. The standard states [ISO/IEC 2011, Section 13.7.123] that the result delivered will be *a processor-dependent approximation* to the value of the l_2 norm; it further recommends that *the processor compute the result without undue overflow or underflow*. It is also expected that the setting of the IEEE arithmetic status flags would be implemented as described under Section 4.5.

Thus the statements

*snrm2 = norm2 (sx(1 : 1+(n-1)*incx : incx))*

and

*dnrm2 = norm2 (dx(1 : 1+(n-1)*incx : incx))*

would be equivalent to a call to the Level 1 BLAS routines *SNRM2* and *DNRM2* respectively when *sx* is a single and *dx* a double precision real array.

The equivalent computation for a complex input vector could be achieved using, for example,

*scnrm2 = norm2([REAL(cx(1:1+(n-1)*incx:incx)), AIMAG(cx(1:1+(n-1)*incx:incx))])*

or

*scnrm2 = norm2(TRANSFER(cx(1:1+(n-1)*incx:incx)),res, SIZE=2*n)*

where *res* is the result of the function and provides the type of the resulting vector.

For the NAG compiler the use of the *TRANSFER* function resulted in a slower overall execution of 50% for single and 20% for double precision input vectors.

To provide all the functionality of the **NRM2* routines we would need to deal with the special cases, $n < 1$ and $incx < 1$, by using wrapper routines. Our tests on this function are effectively restricted to black box testing since we have no access to the sources of the implemented functions.

We tested the *norm2* intrinsic function using the NAG compiler and our test suite. The following observations were noted:

(1) differences in the results obtained for single and double precision arguments may indicate that single precision arguments are cast to double precision, the computation is then performed in double precision and a correctly rounded single precision result is returned:
 (a) for Tests 5 and 6, $x = \{x_1\}$ where $x_1 = HUGE(\cdot)$ and $-HUGE(\cdot)$ respectively, the result returned is exact in both precisions but the inexact flag only signals for double precision arguments
 (b) for Test 23, $x = \{HUGE(\cdot), p\}$ where p is chosen to be the largest power of two such that the returned value is $HUGE(\cdot)$ in the precision being tested. In this case we would also expect the inexact flag to signal. This is what happens in the double precision case but for single precision the function returns $+Inf$ and both the overflow and inexact flags signal. One explanation for this behaviour would be that the computation of the, possibly scaled, sum of squares is per-

formed in double precision and the (double) result is compared to the single precision overflow limit *before* rounding rather than after. Arguments could be made to support both approaches.

(2) contrary to the current Fortran standard [ISO/IEC 2011, 13.7.1], if the result returned is a NaN then the invalid flag does not signal for either precision.

(3) the implementation chooses to differentiate between the two situations discussed in Section 4.5 in which the returned value is $+Inf$.

11. CONCLUSIONS

We have reported on a new Fortran reference implementation of the Level 1 BLAS that compute the l_2 norm of real and complex vectors in both single and double precision. These new codes trade gains in the accuracy in the computed result against execution speed by using compensated summation to collect the sum of squared elements.

The floating point error analysis of the commonly available reference implementations predict a worse case loss of accuracy of $\log_2 n$ bits in the final result where n denotes the number of elements in the input vector. The analysis using compensated summation shows that for $n < N_{max}$, where N_{max} depends on the precision of the floating point arithmetic being used, we should expect close to full precision accuracy in the returned value. In practice, at least for random data in the range $(0, 1)$, we find that we can often exceed N_{max} without loss of full precision accuracy.

For processors that implement the IEEE floating point standard we use intrinsic modules introduced in the Fortran 2003 standard [ISO/IEC 2004] to provide access to the IEEE arithmetic status flags and non-model values. This allows us both to detect and report floating point exceptions that might occur during the computation.

Combining these features with our proposed computational algorithm allows us to provide a reference implementation of **NRM2* that

(1) returns a result that is close to machine accuracy for vector lengths of up to 1.7×10^7 (IEEE single precision) and 9×10^{15} (IEEE double precision),

(2) detects when the result is not storable in the floating point precision being used, sets the return value to Inf, and signals a floating point overflow condition,

(3) detects the presence of non-floating point data (Inf and NaN), returns a suitable result and sets the relevant IEEE status flag,

(4) attempts to balance efficiency with accuracy by employing a hybrid algorithm that uses a fast square and add method without scaling to compute the results and only reverts to a more expensive scaling method if the fast approach causes certain of the IEEE status flags to signal.

The ability to signal exceptions and to return non-finite values means that the routines also provide the user with the opportunity to recover gracefully from a failed computation rather than having the program forcibly terminated by the run-time system.

The new reference implementations are written entirely in standard Fortran 2008 and, thus, in the absence of processor dependent versions of **NRM2*, provide a portable implementation and unambiguous definition of an algorithm that can be

(1) used to replace older versions of the **NRM2* routines to improve accuracy especially for large vectors,

(2) called directly from a supported C compiler using standard conforming C interoperability features available since Fortran 2003 [ISO/IEC 2004],

(3) translated/accessed into/by other languages that require access to a Euclidean norm function.

REFERENCES

E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. 1999. *LAPACK: Users' Guide* (3rd ed.). SIAM, Philadelphia, PA, USA. http://www.netlib.org/lapack/lug/ (Accessed: 2017-09-06).

ANSI. 1966. *Programming Language Fortran X3.9-1966*. American National Standards Institute, New York.

D. H. Bailey. 1995. A Fortran 90-based Multiprecision System. *ACM Trans. Math. Software* 21, 4 (Dec. 1995), 379–387. DOI:http://dx.doi.org/10.1145/212066.212075

J. L. Blue. 1978. A Portable Fortran Program to Find the Euclidean Norm of a Vector. *ACM Trans. Math. Software* 4, 1 (1978), 15–23. DOI:http://dx.doi.org/10.1145/355769.355771

J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. 1990. Algorithm 679: a Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software* 16, 1 (March 1990), 18–28. DOI:http://dx.doi.org/10.1145/77626.77627

J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. 1988. Algorithm 656: an Extended Set of Basic Linear Algebra Subprograms: Model Implementation and Test Programs. *ACM Trans. Math. Software* 14, 1 (March 1988), 18–32. DOI:http://dx.doi.org/10.1145/42288.42292

J. J. Dongarra and E. Grosse. 1987. Distribution of Mathematical Software via Electronic Mail. *Comm. ACM* 30, 5 (May 1987), 403–407. DOI:http://dx.doi.org/10.1145/22899.22904

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

S. Graillat, C. Lauter, P. Tang, N. Yamanaka, and S. Oishi. 2015. Efficient Calculations of Faithfully Rounded L2-Norms of n-Vectors. *ACM Trans. Math. Software* 41, 4, Article 24 (Oct. 2015), 20 pages. DOI:http://dx.doi.org/10.1145/2699469 Software package available from http://www.christoph-lauter.org/faithfulnorm.tgz (Accessed: 2017-09-06).

R. J. Hanson and T. R. Hopkins. 2015. Remark on Algorithm 539: A Modern Fortran Implementation for Carefully Computing the Euclidean Norm — User Manual. (2015). Manual accompanying the software component of this publication.

R. J. Hanson and F. T. Krogh. 1987. Algorithm 653: Translation of Algorithm 539: PC-BLAS Basic Linear Algebra Subprograms for FORTRAN Usage with the INTEL 8087 80287 Numeric Data Processor. *ACM Trans. Math. Software* 13, 3 (Sept. 1987), 311–317. DOI:http://dx.doi.org/10.1145/29380.214346

Y. Hida, X. S. Li, and D. H. Bailey. 2007. Library for Double-Double and Quad-Double Arithmetic. (Dec. 2007). http://crd-legacy.lbl.gov/~dhbailey/mpdist/qd-2.3.17.tar.gz (Accessed 2017-09-06).

N. J. Higham. 2002. *Accuracy and Stability of Numerical Algorithms* (second ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA. xxx+680 pages.

T.R. Hopkins. 1996. Restructuring Software: A Case Study. *Software — Practice and Experience* 26, 8 (Aug. 1996), 967–982. DOI:http://dx.doi.org/10.1002/(SICI)1097-024X(199608)26:8⟨967::AID-SPE41⟩3.0.CO;2-G

ISO/IEC. 1991. *Information Technology – Programming Languages – FORTRAN (ISO/IEC 1539:1991)*. ISO/IEC Copyright Office, Geneva, Switzerland.

ISO/IEC. 1997. *Information Technology – Programming Languages – Fortran – Part 1: Base Language (ISO/IEC 1539-1:1997)*. ISO/IEC Copyright Office, Geneva, Switzerland.

ISO/IEC. 2004. *Information Technology – Programming Languages – Fortran – Part 1: Base Language (ISO/IEC 1539-1:2004)*. ISO/IEC Copyright Office, Geneva, Switzerland.

ISO/IEC. 2011. *Information Technology – Programming Languages – Fortran – Part 1: Base Language (ISO/IEC 1539-1:2010)*. ISO/IEC Copyright Office, Geneva, Switzerland.

W. Kahan. 1997. *Lecture Notes on the IEEE Standard 754 for Binary Floating-Point Arithmetic*. Technical Report. Electrical Engineering and Computer Sciences, University of California, California, US. http://www.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF (Accessed: 2017-09-06).

W. Kahan. 2012. Private communication. (2012).

C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979a. Algorithm 539: Basic Linear Algebraic Subprograms for Fortran Usage. *ACM Trans. Math. Software* 5, 3 (Sept. 1979), 324–325. DOI:http://dx.doi.org/10.1145/355841.355848

C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979b. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Software* 5, 3 (Sept. 1979), 308–323. DOI:http://dx.doi.org/10.1145/355841.355847

M. Metcalf, J. Reid, and M. Cohen. 2011. *Modern Fortran Explained*. Oxford University Press, Oxford, UK.

MPFR. 2016. The GNU MPFR (Multiple Precision Floating Point Reliable) Library Download Site. (2016). http://www.mpfr.org/ (Accessed: 2017-09-06).

IEEE Task P754. 1985. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York. A preliminary draft was published in the January 1980 issue of IEEE Computer, together with several companion articles.

LAPACK Project. 2015. LAPACK Official Download Site. (2015). http://www.netlib.org/lapack/ (Accessed: 2017-09-06).

Received xxx 2016; revised ; accepted