

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

Richard J. Hanson      Tim Hopkins

January 18, 2017

## Abstract

This document contains details of the software package that accompanies the article *Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm*.

A description is provided of each test that comprises the extensive test suite designed to test extensively modern implementations of the BLAS Level 1 *\*nrm2* routines for computing the  $l_2$  norm of a real or complex vector. Details are given to allow the user to add more tests if required.

A benchmark package is described which compares the performance (execution time) and accuracy of the proposed method against a competitive implementation. A number of alternative algorithms/implementations are provided including the original BLAS Level 1, the implementation that accompanies the Lapack package, Blue’s algorithm and a number of variants of these methods that use compensated summation and more accurate scaling. The format of the optional data file that may be supplied to alter the benchmark parameters is also described.

Details are given of the two makefiles that accompany the package along with information on how to edit the *makefile.inc* file for a new platform and the requirements on the Fortran compiler to build the testing and benchmark executables.

Information is also provided on how to use the test software and the benchmark program with a new implementation of the *\*nrm2* routines.

# 1 Introduction

This user manual accompanies the software implementing a new hybrid algorithm for computing the  $l_2$  norm of a real or complex  $n$ -vector which is described in the article [6].

Section 2 contains detail of the individual tests that form part of the comprehensive test suite which is suitable for testing any set of routines with the same structure as the BLAS Level-1 *\*nrm2* routines. A benchmark program that compares the performance of competing algorithms and implementations from both a timing and an accuracy viewpoint is described in Section 3.

In Sections 4 and 5 we describe the structure and contents of the files which make up the software package and the accompanying makefiles. Finally in Section 6 we provide details of the changes needed

1. to add new tests to the test suite, and
2. to allow a new implementation of the  $l_2$  norm routines to be tested against the test suite and benchmarked against the proposed hybrid code.

# 2 Testing

By using a combination of black box and white box testing techniques, we have produced a comprehensive set of tests that we believe would prove extensive for any implementation of a routine designed to compute the  $l_2$  norm of a real or complex vector. All the following implementations have been tested using the complete suite

1. the original Level 1 Blas reference routines [11],
2. the reference LAPACK [1] implementation,
3. Kahan's algorithm in stand-alone mode,
4. our proposed hybrid implementation.

The tests have been devised to make the computed results automatically checkable.

The tests given below are all presented as real data. The same data is used for testing the complex routines after transforming it as follows:

*incx* = 1 If the number of elements,  $n$ , in the real input vector,  $x$ , is even, then we generate a complex input vector,  $xc$ , of length  $n/2$  where  $xc(j) = CMPLX(x(2*j-1), x(2*j))$  for  $j = 1, \dots, n/2$ . If  $n$  is odd then the final element is  $xc(\lceil n/2 \rceil) = CMPLX(x(n), zero)$ .

*incx* > 1 Here we generate a complex input vector,  $xc$ , of the same length as the real input vector,  $n$ , with  $xc(i)$  set to  $CMPLX(x(i), zero)$  or  $CMPLX(zero, x(i))$  depending on whether  $i$  is odd or even respectively.

Using this transformation the results for the real and complex data should be identical.

Table 1 gives values for the parameters used later in the paper to define some of the test data sets. Numerical values are given for both single and double precision IEEE standard formats.

The IEEE floating point status flags signalling on return from any subprogram call must be the union of the flags signalling on entry to the subprogram and those set within the routine itself. When the input vector consists entirely of finite floating-point values, the only flags that may be set as a result of a call to a *\*nrm2* routine are overflow and inexact. The overflow flag should then only signal when the value of the final result is not representable in the precision being used; in this case the inexact flag should also signal. If one of the elements of the input vector is  $\pm Inf$  then the result is exactly  $\pm Inf$  and no overflow flag is set. The invalid flag will signal if a NaN value (but no Inf values) are detected in the input vector.

Note that the underflow condition should never occur from a call to a *\*nrm2* routine since  $\|x\|_2 \geq |x_i| > 0$  provided the vector,  $x$ , consists entirely of finite floating-point values with at least

Table 1: Values of the constants used in setting test data values.

	Single Precision	Double Precision
$\omega$	$(2 - 2^{-23})2^{127} \approx 2^{128}$	$(2 - 2^{-52})2^{1023} \approx 2^{1024}$
$\tau$	$2^{-126}$	$2^{-1022}$
$\epsilon$	$2^{-23}$	$2^{-52}$
$\Omega$	128	1024
$\eta$	-23	-52
$\alpha$	-11	-26
$\gamma$	-125	-1021
$\delta$	24	53
Key		
$\omega$	HUGE(kind)	
$\tau$	TINY(kind)	
$\epsilon$	EPSILON(kind)	
$\Omega$	EXPONENT(HUGE(kind))+1	
$\eta$	EXPONENT( $\epsilon$ )-1	
$\alpha$	$\lfloor \epsilon/2 \rfloor$	
$\gamma$	MINEXPONENT(kind)	
$\delta$	DIGITS(kind)	
<i>NOTE: the definition of <math>\omega</math> and <math>\tau</math> used here differ from those given in the main article.</i>		

one element non-zero. Thus, while an intermediate computation could underflow if, for example, no scaling were employed, the final correct result can only be zero if all the elements of the vector are zero.

Any test failures that involve the inexact flag (element 5 in the output vector of status flags) need to be considered carefully as the result may vary depending upon the algorithm used. Many of the test vectors contain elements that are either small integers or small integers scaled by a power of two with a resultant  $l_2$  norm that is exactly representable in IEEE floating point format. In these cases we do not expect the inexact status flag to be raised. This is in line with the goal we set ourselves in [6] although it does favour algorithms that use scaling factors that are powers of two since division and multiplication by such values should not generate any rounding errors.

However some test cases may or may not signal inexact depending on how the calculations are performed. One example of this is discussed below for vectors of length one. A second more general example is the vector  $x = \{9 * p, 11 * p, 13 * p, 23 * p\}$  where  $p$  is a power of two which is large enough to cause the implementation to scale. Provided the result does not overflow (i.e.,  $p$  is small enough) we would expect that the exact result,  $30 * p$ , will be returned. However, if any of the chosen scale factors is not a power of two then the resultant scaling/rescaling will generate rounding errors and an inexact result will ensue. This will happen using both the original BLAS implementation and the Lapack routine; details of both routines may be found in [6].

When the  $l_2$  norm cannot be represented exactly using IEEE floating point format, the mathematically correct result generally involves the square root of an integer value multiplied by a power of two. When computing relative errors we assume that the Fortran intrinsic function, SQRT, returns a correctly rounded result which may be used to generate the best available approximation to the exact value. Deviations from this value then signal additional rounding errors which may be due, for example, to poor scaling.

## 2.1 Tests for *\*nrm2* Routines

We can split the input domain into a number of ‘obvious’ classes

1. Inputs which generate a representable floating point value,
2. Inputs which generate a non-representable floating-point value,
3. Input data that includes non-finite values,
4. Input data that includes not-a-number values,
5. Trivial input data,
6. Illegal input data.

We start with classes 5 and 6 since these are small classes. As we are aiming to deal with any data within the input vector, i.e., both valid floating-point numbers as well as infinite and NaN values, the only illegal data is obtained by setting *incx* to be less than one. The only trivial data is the case of a null array which we assume is being signaled if the value of *n* is set to be non-positive. In both cases we choose to return the value zero for the norm in line with previous implementations. Thus our first two sets of test data are

**Test 1**  $n = 0, incx = 1, x = \{\} \Rightarrow result = 0.0$

**Test 2**  $n = 1, incx = 0, x = \{\} \Rightarrow result = 0.0$

We choose *n* and *incx* to be boundary values; i.e., they are values closest to legal values.

$n = 1$  could be treated as a special case although this is very unlikely to provide any efficiency gain since it is not a common practical occurrence. Treating this as a special case for real input data could cause an inconsistency with the setting of the IEEE status flags; for example, consider  $x = \omega$  using the proposed hybrid method, then

1. treating this as a special case would give a result of  $\omega$  with the inexact flag not triggered,
2. not treating it as a special case would generally generate the same result but the inexact flag would be set due to the computation within Kahan’s algorithm (two scalings, a square and a square root) on a number which is not an exact power of two,
3.  $n = 1$  is not treated as a special case for complex input where it is equivalent to a real vector of length two with  $x = \{\omega, zero\}$  and this causes the inexact flag to be set as in 2 above.

Finally, a vector of length *n* where  $x_i = \omega$  and  $x_j = zero$  for  $j \neq i$  will also cause the inexact flag to be set.

We therefore choose not to treat  $n = 1$  as a special case (resulting in a generally irrelevant loss of accuracy for real vectors of length one!). We do, however, include a set of tests with  $n = 1$  as these provide

1. simple tests for our software, and
2. tests for an implementation that does treat  $n = 1$  as a special case.

We also include a test (Test 14) of the form  $\{0, 0, \dots, 0, \omega, 0, \dots, 0\}$  to check for consistency (or to detect  $n = 1$  being treated as special).

**Test 3**  $n = 1, incx = 1, x = \{3.0\} \Rightarrow result = 3.0$

**Test 4**  $n = 1, incx = 1, x = \{-3.0\} \Rightarrow result = 3.0$

The *ieee\_inexact* status flag is not set in the next four tests (5–8) as the exact result can be represented; this will happen if  $n = 1$  is treated as a special case or the method uses  $x_1$  as a scale factor.

**Test 5**  $n = 1, incx = 1, x = \{\omega\} \Rightarrow result = \omega$

**Test 6**  $n = 1, incx = 1, x = \{-\omega\} \Rightarrow result = \omega$

**Test 7**  $n = 1, incx = 1, x = \{\tau\} \Rightarrow result = \tau$

**Test 8**  $n = 1, incx = 1, x = \{-\tau\} \Rightarrow result = \tau$

**Test 9**  $n = 1, incx = 1, x = \{+Inf\} \Rightarrow result = +Inf$

**Test 10**  $n = 1, incx = 1, x = \{-Inf\} \Rightarrow result = +Inf$

**Test 11**  $n = 1, incx = 1, x = \{sNaN\} \Rightarrow result = qNaN$  (*ieee\_invalid set*)

**Test 12**  $n = 1, incx = 1, x = \{qNaN\} \Rightarrow result = qNaN$  (*ieee\_invalid set*)

Check that there are no problems if the algorithm scales and the input vector contains only zero elements:

**Test 13**  $n = 10, incx = 1, x = \{0, 0, \dots, 0, 0\} \Rightarrow result = 0$

Test for vector of all zeros except for a single element of value  $\omega$ :

**Test 14**  $n = 5, incx = 1, x = \{0, 0, \omega, 0, 0\} \Rightarrow result = \omega$  (*the comments preceding Test 5 also apply here*)

For the two classes 3 and 4 we choose an input vector that contains infinite or NaN values. For completeness we use both positive and negative infinity along with signaling and quiet NaN values:

**Test 15**  $n = 3, incx = 1, x = \{2.0, -1.0, +Inf\} \Rightarrow result = +Inf$

**Test 16**  $n = 3, incx = 1, x = \{2.0, -Inf, -1.0\} \Rightarrow result = +Inf$

**Test 17**  $n = 3, incx = 1, x = \{2.0, -Inf, sNaN\} \Rightarrow result = +Inf$

**Test 18**  $n = 3, incx = 1, x = \{2.0, sNaN, -1.0\} \Rightarrow result = qNaN$  (*ieee\_invalid set*)

**Test 19**  $n = 3, incx = 1, x = \{qNaN, 2.0, -1.0\} \Rightarrow result = qNaN$  (*ieee\_invalid set*)

**Test 20**  $n = 4, incx = 1, x = \{qNaN, sNaN, -Inf, sNaN\} \Rightarrow result = +Inf$

For class 2 we need to consider a vector of size at least two since, for  $n = 1$ , if the element is a representable floating-point value, then we can represent the value of the  $l_2$  norm as it is just the absolute value of the input and all valid negative IEEE values have a valid positive value of the same magnitude. Hence, an obvious test would be:

**Test 21**  $n = 2, incx = 1, x = \{\omega, \omega\} \Rightarrow result = +Inf$  (*ieee\_overflow and ieee\_inexact set*)

The next two data sets use boundary values to test that the routine returns a finite value/overflows at the expected point. Use  $x = \{\omega, x_2\}$  and choose  $x_2$  to be the largest value which won't cause an overflow and the smallest value which will using floating-point arithmetic. We have

$$res = \sqrt{\omega^2 + (x_2)^2} = \omega \sqrt{\left(1 + \frac{x_2^2}{\omega^2}\right)} \quad (1)$$

If the software is performing to its specification, we should obtain an infinite result only if  $(1 + x_2^2/\omega^2) > 1$  otherwise a value of  $\omega$  should be returned. Thus an unrecoverable overflow would occur if  $x_2^2 \geq \omega^2 \epsilon$  and using  $\omega = 2^\Omega$  to obtain an exact power of two gives  $x_2 \geq 2^{\Omega+\alpha}$ .

Whence,

**Test 22**  $n = 2$ ,  $incx = 1$ ,  $x = \{\omega, 2^{\Omega+\alpha}\} \Rightarrow result = +Inf$  (*ieee\_overflow and ieee\_inexact set*)

**Test 23**  $n = 2$ ,  $incx = 1$ ,  $x = \{\omega, 2^{\Omega+\alpha-1}\} \Rightarrow result = \omega$  (*ieee\_inexact set*)

We should also check that we do not have any problems with underflows:

**Test 24**  $n = 4$ ,  $incx = 1$ ,  $x = \{\tau, \tau, \tau, \tau\} \Rightarrow result = 2 * \tau$

Next we consider inputs for which the sum of squares of the elements is exactly representable as an IEEE floating point value.

The first test may cause the inexact flag to be set if non-powers of two are used for scaling, however, the exact result is representable. The second test should set the inexact flag and should agree with the value provided by the Fortran square root intrinsic.

**Test 25**  $n = 3$ ,  $incx = 1$ ,  $x = \{9.0, 18.0, 38.0\} \Rightarrow result = 43.0$

**Test 26**  $n = 3$ ,  $incx = 1$ ,  $x = \{10.0, 11.0, 12.0\} \Rightarrow result = \sqrt{365}$  (*ieee\_inexact set*)

It would also be useful to have an example for a larger value of  $n$ . Hirschhorn [7] provides the following

$$\sum_{i=0}^{k-1} (a+i)^2 = \sum_{i=0}^{k-1} r_i^2 = b^2 \quad (2)$$

where  $k = v^2$ ,  $a = (v^4 - 24v^2 - 25)/48$  and  $v$  is any integer not divisible by 2 or 3. The required sum is then  $b^2$  where  $b$  is given by

$$b = (v^5 + 47v)/48 \quad (3)$$

As  $v$  increases both the starting value,  $a$ , and the number of terms also increase. We note that all the values are integer and, hence, for smaller values of  $v$  the sum of squares will be collected exactly. We denote the largest value of  $v$  that allows an exact result to be computed in the chosen precision by  $v_1$ ; the next valid value of  $v = v_2$  should generate an inexact result. For IEEE single and double precision the values for  $(v_1, v_2)$  are  $(11, 13)$  and  $(85, 89)$  respectively. The subroutine `getVals` in the module `sdTestMod` attempts to compute these values automatically. Hence

**Test 27**  $n = v_1^2$ ,  $incx = 1$ ,  $x = \{r_0, r_1, \dots, r_{n-1}\}$  from equation (2)  $\Rightarrow result = 3366.0$  and  $92438690.0$  for IEEE single and double precision respectively.

would provide an extended test. We should also add

**Test 28**  $n = v_2^2$ ,  $incx = 1$ ,  $x = \{r_0, r_1, \dots, r_{n-1}\}$  from equation (2)  $\Rightarrow result = 7748.0$  and  $116334659.0$  for IEEE single and double precision respectively. (*ieee\_inexact set*)

to ensure that the IEEE flags are set correctly on exit.

The following six tests would all cause the simple loop to overflow or underflow. The first two should generate the exact result whether or not the scaling factors used are a power of two; the second pair also have an  $l_2$  norm which is exactly representable but using scaling factors that are not powers of two will probably lead to the inexact flag signalling.

**Test 29**  $n = 4$ ,  $incx = 1$ ,  $x = \{p, p, p, p\} \Rightarrow result = 2 * p$

**Test 30**  $n = 4$ ,  $incx = 1$ ,  $x = \{q, q, q, q\} \Rightarrow result = 2 * q$

**Test 31**  $n = 4$ ,  $incx = 1$ ,  $x = \{9 * p, 11 * p, 13 * p, 23 * p\} \Rightarrow result = 30 * p$

**Test 32**  $n = 4$ ,  $incx = 1$ ,  $x = \{9 * q, 11 * q, 13 * q, 23 * q\} \Rightarrow result = 30 * q$

The last pair of tests should both generate results with the inexact flag set due to the square root operation

**Test 33**  $n = 5, incx = 1, x = \{p, p, p, p, p/4\} \Rightarrow result = \sqrt{65}p/4$  (*ieee\_inexact set*)

**Test 34**  $n = 5, incx = 1, x = \{q, q, q, q, 2 * q\} \Rightarrow result = 2\sqrt{2}q$  (*ieee\_inexact set*)

where  $p = 2^{\Omega/2-1}$  and  $q = 2^{-\Omega/2}$ .

In the next test the squaring would result in an unnormalized number, if they are available, or zero, if they aren't. Either way an underflow would occur with the simple loop. Kahan's algorithm will use scaling to generate an inexact, but accurate, result.

**Test 35**  $n = 4, incx = 1, x = \{p, p, p, p\} \Rightarrow result = 2 * p$  where  $p = r^{-\Omega/2}$  and  $r$  is the nearest floating point number less than one. (*ieee\_inexact set*)

The following two tests are designed to show up scaling problems within the elements of the vector. The value of  $p$  is chosen so that  $1 + p^2$  returns 1 under the floating point arithmetic being used. The vector is then made large enough so that  $1 + \sum p^2$  would be representable. Using a naive approach the first test will return 1 while a similar approach to the second should provide an accurate result.

**Test 36**  $n = 2^{nval} + 1, incx = 1, x = \{1, p, p, p, \dots, p\} \Rightarrow result = \sqrt{1 + p^2 * 2^{nval}}$  (*ieee\_inexact set*)

**Test 37**  $n = 2^{nval} + 1, incx = 1, x = \{p, p, p, \dots, p, 1\} \Rightarrow result = \sqrt{1 + p^2 * 2^{nval}}$  (*ieee\_inexact set*)

where  $p = 2^{\alpha-2}$  and  $nval = 10$ .

The next tests check that the implementation operates correctly when  $incx \neq 1$  on a series of simple examples.

**Test 38**  $n = 3, incx = 2, x = \{2.0, 5.0, 3.0, 9.0, 6.0\} \Rightarrow result = 7.0$

**Test 39**  $n = 2, incx = 3, x = \{2.0, 3.0, 4.0, 5.0\} \Rightarrow result = \sqrt{29}$  (*ieee\_inexact set*)

**Test 40**  $n = 3, incx = 2, x = \{2.0, +Inf, 3.0, qNaN, 6.0\} \Rightarrow result = 7.0$

The next two tests are algorithm specific and attempt to check that the different accumulator elements are combined correctly; to be specific: medium and big; and small and medium.

**Test 41**  $n = 2, incx = 1, x = \{2^{c-1}, 2^{c+1}\} \Rightarrow result = \sqrt{17} \times 2^{c-1}$  (*ieee\_inexact set*)

**Test 42**  $n = 2, incx = 1, x = \{2^{d-1}, 2^{d+1}\} \Rightarrow result = \sqrt{17} \times 2^{d-1}$  (*ieee\_inexact set*)

where  $B = f_B 2^c$  and  $b = f_b 2^d$  are defined in Blue's algorithm and  $c = EXPONENT(B)$ ,  $d = EXPONENT(b)$ ,  $f_B = FRACTION(B)$  and  $f_b = FRACTION(b)$  as provided by the Fortran 2008 elemental functions with  $f_b, f_B \in [\frac{1}{2}, 2)$  for IEEE floating point arithmetic.

The next six tests check that the IEEE status flags are processed correctly. We have tested that the flags are set correctly in the above tests when the input status flags are all set to FALSE on entry. The following tests mostly check that when the input flags have all been set to TRUE then the flags are all still TRUE on exit whether the call to *\*nrm2* would have signalled or not.

In these six tests the status flag settings are stored in a logical array of length five in the order

$\{ieee\_overflow, ieee\_divide\_by\_zero, ieee\_invalid, ieee\_underflow, ieee\_inexact\}$

We give the definition of the test problem and the state of the input flags along with the expected result and exit settings for the flags.

**Test 43**  $n = 2$ ,  $incx = 1$ ,  $x = \{5, 12\}$ ,  $status = \{T, T, T, T, T\} \Rightarrow result = 13$ ,  $status = \{T, T, T, T, T\}$

**Test 44**  $n = 2$ ,  $incx = 1$ ,  $x = \{7, 9\}$ ,  $status = \{T, T, T, T, T\} \Rightarrow result = \sqrt{130}$ ,  $status = \{T, T, T, T, T\}$

**Test 45**  $n = 2$ ,  $incx = 1$ ,  $x = \{7, 9\}$ ,  $status = \{T, T, T, T, F\} \Rightarrow result = \sqrt{130}$ ,  $status = \{T, T, T, T, T\}$

In the next three tests  $p = 2^{\Omega/2-1}$

**Test 46**  $n = 4$ ,  $incx = 1$ ,  $x = \{p, p, p, p\}$ ,  $status = \{T, T, T, T, T\} \Rightarrow result = 2 * p$ ,  $status = \{T, T, T, T, T\}$

**Test 47**  $n = 5$ ,  $incx = 1$ ,  $x = \{p, p, p, p, p/4\}$ ,  $status = \{T, T, T, T, T\} \Rightarrow result = \sqrt{65}p/4$ ,  $status = \{T, T, T, T, T\}$

**Test 48**  $n = 5$ ,  $incx = 1$ ,  $x = \{p, p, p, p, p/4\}$ ,  $status = \{T, T, T, T, F\} \Rightarrow result = \sqrt{65}p/4$ ,  $status = \{T, T, T, T, T\}$

Finally, these tests require the floating-point system to provide gradual underflow and they check that underflows are not triggered for input vectors containing elements in the gradual underflow range.

**Test 49**  $n = 4$ ,  $incx = 1$ ,  $x = \{p, p, p, p\} \Rightarrow result = 2 * p$  where  $p = \tau/4$

**Test 50**  $n = 4$ ,  $incx = 1$ ,  $x = \{p, p, p, p\} \Rightarrow result = 2 * p$  where  $p$  is the smallest representable real value  $= 2^{\gamma-\delta}$

**Test 51**  $n = 4$ ,  $incx = 1$ ,  $x = \{p, p, p, p\} \Rightarrow result = 2 * p$  where  $p$  is the next representable real value  $> \tau$

## 2.2 Output of IEEE arithmetic status flags

The package checks and reports on the final settings of the five IEEE floating-point status flags that can be set during the execution of the test suite:

1. overflow
2. divide by zero
3. invalid
4. underflow
5. inexact

and outputs the values as a string of five characters separated by spaces. The order of the characters in the output is the same as the order of the flags given above.

Each flag may take the value

**T**: status flag has signalled during the computation of the norm,

**F**: status flag has not signalled,

**X**: the status flag is not supported by the compiler/platform combination being used.

For example, the string *T\_F\_X\_T\_F* would signify that



Test No.	n	incx	Input			Result	Status Flags		
			fp	Inf	NaN		Overflow	Invalid	Inexact
1	0	1	—	—	—	zero	—	—	—
2	1	0	—	—	—	zero	—	—	—
3	1	1	×	—	—	exact fp	—	—	—
4	1	2	×	—	—	exact fp	—	—	—
5	1	1	×	—	—	exact fp	—	—	?
6	1	1	×	—	—	exact fp	—	—	?
7	1	1	×	—	—	exact fp	—	—	?
8	1	1	×	—	—	exact fp	—	—	?
9	1	1	—	×	—	<i>Inf</i>	—	—	—
10	1	1	—	×	—	<i>Inf</i>	—	—	—
11	1	1	—	—	<i>q</i>	<i>NaN</i>	—	×	—
12	1	1	—	—	<i>s</i>	<i>NaN</i>	—	×	—
13	10	1	×	—	—	zero	—	—	—
14	5	1	×	—	—	exact fp	—	—	?
15	3	1	×	×	—	<i>Inf</i>	—	—	—
16	3	1	×	×	—	<i>Inf</i>	—	—	—
17	3	1	×	×	<i>s</i>	<i>Inf</i>	—	—	—
18	3	1	×	—	<i>s</i>	<i>NaN</i>	—	×	—
19	3	1	×	—	<i>q</i>	<i>NaN</i>	—	×	—
20	4	1	—	×	<i>qs</i>	<i>Inf</i>	—	—	—
21	2	1	×	—	—	<i>Inf</i>	×	—	×
22	2	1	×	—	—	<i>Inf</i>	×	—	×
23	2	1	×	—	—	inexact fp	—	—	×
24	4	1	×	—	—	exact fp	—	—	—
25	3	1	×	—	—	exact fp	—	—	—
26	3	1	×	—	—	inexact fp	—	—	×
27	s/d	1	×	—	—	exact fp	—	—	—
28	s/d	1	×	—	—	inexact fp	—	—	×
29	4	1	×	—	—	exact fp	—	—	—
30	4	1	×	—	—	exact fp	—	—	—
31	4	1	×	—	—	exact fp	—	—	—
32	4	1	×	—	—	exact fp	—	—	—
33	5	1	×	—	—	inexact fp	—	—	×
34	5	1	×	—	—	inexact fp	—	—	×
35	4	1	×	—	—	inexact fp	—	—	×
36	1025	1	×	—	—	inexact fp	—	—	×
37	1025	1	×	—	—	inexact fp	—	—	×
38	3	2	×	—	—	exact fp	—	—	—
39	2	3	×	—	—	inexact fp	—	—	×
40	3	2	×	<i>n/a</i>	<i>n/a</i>	exact fp	—	—	—
41	2	1	×	—	—	inexact fp	—	—	×
42	2	1	×	—	—	inexact fp	—	—	×
43	2	1	×	—	—	exact fp	—	—	—
44	2	1	×	—	—	inexact fp	—	—	×
45	2	1	×	—	—	inexact fp	—	—	×
46	4	1	×	—	—	exact fp	—	—	—
47	5	1	×	—	—	inexact fp	—	—	—
48	5	1	×	—	—	inexact fp	—	—	—
49	4	1	<i>d</i>	—	—	exact fp	—	—	—
50	4	1	<i>d</i>	—	—	exact fp	—	—	—
51	4	1	<i>d</i>	—	—	inexact fp	—	—	×

Table 2: Table showing details of test input data, expected results and expected status flag settings

- the overflow and underflow flags signalled,
- the divide by zero and inexact flags did not signal, and
- the invalid flag was reported as not being supported.

If any of the characters used in the string offends the user they may be changed by altering the values of *chTrue* (flag signalling), *chFalse* (flag not signalling) and *chUnsupported* (flag not supported) in the module variables section of the module *chTests*.

A discussion of the testing and reporting of the inexact flag may be found at the start of this section.

## 2.3 Controlling computation and output of the test suite

The user may control

1. the amount of detail output when running the test suite, and
2. what results are compared and reported.

Different controls may be applied independently to the single and double precision versions of the test suite.

The level of output obtained may be altered using the routine *setOutputLevel* from the module *checkTests*. This routine takes a single integer argument *level*; for example

```
CALL setOutputLevel(2)
```

sets the output level to 2. The level set remains in force throughout the execution of the test suite unless it is altered by a subsequent call. Thus we may obtain different levels of output for the different precisions using

```
! Set output level for double precision tests
CALL setOutputLevel(2)
! ... double precision test calls ...

CALL setOutputLevel(1)
! ... single precision test calls ...
```

The default level is *level* = 3 and this level will be used if no calls to *setOutputLevel* have been made prior to the test suite being executed. If the value provided to *setOutputLevel* is outside of the range [1,3] then the value 3 is used. The current value of *level* may be obtained using the function *getOutputLevel* which returns the value as an integer:

```
currentLevel = getOutputLevel()
```

The different levels available are

- *level* = 1: full output of all test results.

Results are printed for each individual test and for both real and complex input vectors whether or not the test fails. The output includes

- the title of the test,
- the type of the input vector (*REAL* or *COMPLEX*),
- the computed  $l_2$ -norm,
- the returned IEEE arithmetic status flags,
- a count of the number of failures reported.

For both the results and the status flags the program also outputs either *Correct* or *\*\*\*Incorrect\*\*\** tags after comparing the computed values with the expected values. If any of the computed results do not agree, the expected results are also output along with a ‘banner’ output line. For example:

```
Test 22: Boundary value causing overflow
Using REAL input vector
Result returned =          +Infinity  Correct
IEEE status flags returned:      T F F F T **Incorrect**
                                expected:  F F F F T
***** TEST FAILED *****
```

In the case where the computational result differs from the expected, a line giving the expected value and the relative difference

$$\left| \frac{\text{expected} - \text{computed}}{\text{expected}} \right|$$

is output unless the expected output is zero when the value

$$|\text{computed}|$$

is used.

- *level = 2*: compressed output of all test results.

As with *level = 1* above this level produces output for all the test cases but uses a more compact form from the detailed level. If the computed results agree with the expected then a single line is output and tagged *Correct*. If any differences occur then, in the case of the computed result, the error in the returned value is given (as detailed in *level = 1* above). When the IEEE status flags do not agree then the expected flags are output directly underneath the returned values. In both cases the incorrect test output is highlighted by using the tag *\*\*\*Incorrect\*\**. A count of the number of failures is also reported.

For example:

```
22 R          +Infinity      NONE      T F F F T **Incorrect**
                                F F F F T
```

- *level = 3*: compressed output of unexpected test results.

The output is the same as *level = 2* above but only occurs if differences are detected between the computed and expected results. A count of the number of failures is also reported.

## 2.4 Controlling the scope of the testing

By default the test driver will check both the return values of the  $l_2$  norm and the settings of the IEEE status flags. It is also possible just to check the returned values and ignore the status flags in both the test and the report.

The routine *setJustValues* from the module *checkTests* has the form

```
CALL setJustValues(value)
```

where *value* is a simple logical variable. If *value* is set to *.TRUE.* then only the computed values are compared and reported. If *value* is set to *.FALSE.* (the default) then both computed values and the IEEE status flags are compared and reported.

The current setting may be retrieved using

```
value = getJustValues()
```

## 2.5 The datatype *testInfoType*

The data type *testInfoType* is used to store the definition of a test, the expected computed value and returned IEEE status flags, the returned computed value and the returned IEEE status flags, and a title/short description of the test which is available for output. It is defined as follows:

```

TYPE testInfoType
  INTEGER :: testNo
  CHARACTER(LEN=:), ALLOCATABLE :: testDescript
  REAL(wp), POINTER :: inVector(:)
  INTEGER :: nVal, incxVal
  LOGICAL :: initialStatus(5)
  REAL(wp) :: expectedNorm
  LOGICAL :: expectedStatus(5)
  REAL(wp) :: returnedNorm
  LOGICAL :: returnedStatus(5)
END TYPE testInfoType

```

A short description of each component follows:

***testNo*** is the test number associated with the test. This is used to identify the particular test in the output. Details of how to add extra tests to the suite given in Section 2.6.

***testDescript*** contains a short description of the test and is output for some levels of output.

The next three components define the data to be used in the test. The call to an *\*nrm2* routine is of the form

```
returnedNorm = *nrm2(nVal, inVector, incxVal)
```

***inVector*** is the data for the real array – for details of how the equivalent complex vector is generated see the start of this section.

***nVal*** is the number of elements to be used in computing the norm,

***incxVal*** is the gap to be used between elements; elements used are  $(1, incxVal + 1, 2 * incxVal + 1, \dots, (nVal - 1) * incxVal + 1)$ ,

***initialStatus*** defines the IEEE status flags on entry to the *\*nrm2* routine. These may be used for two purposes:

1. set to all FALSE to test that the *\*nrm2* implementation under test only signals the correct flags,
2. The flags can also be set to test that the correct union of flags is returned after executing the *\*nrm2* routine, i.e., the call to the *\*nrm2* routine does not adversely affect any existing settings.

***expectedNorm*** defines the expected return value from the call to the *\*nrm2* routine. This will be the same value for the real and complex calls for a given precision. This should be the correctly rounded representation of the exact value.

***expectedStatus*** defines the expected values of the IEEE status flags on exit from the *\*nrm2* routine. This should take account of the initial setting.

***returnedNorm*** is the computed value returned by the call to the *\*nrm2* routine.

***returnedStatus*** is the vector of IEEE status flags following the call to the *\*nrm2* routine.

## 2.6 Adding a New Test Case

The code defining the individual test cases within the test suite is linear – each test being executed in its order of appearance within the file *sdTestMod.inc*.

The test number assigned to the test depends on its order within the sequence. Thus adding a new test in the middle of the existing suite will move all the following test numbers out of alignment with the descriptions in Section 2. It is, therefore, recommended that additional tests are added after the comment

```
! Any new tests should be added here in the first instance to
! prevent misaligning test numbers with the descriptions given
! in the user manual.
```

just before the definition of *SUBROUTINE getVals* in the file *sdTestMod.inc*.

### 2.6.1 Format of the Test Definition

In the following, names like *nval*, *incxVal*, etc are shortened forms of the components of the *testInfoType* defined via the *ASSOCIATE* statement:

```
ASSOCIATE(testNo=>testInfo%testNo, &
          expStat=>testInfo%expectedStatus, &
          expRes =>testInfo%expectedNorm, &
          nVal => testInfo%nVal, &
          incxVal => testInfo%incxVal, &
          initStat => testInfo%initialStatus)
```

The components *inVector* and *testdescript* are dynamic and are quoted in full.

We use an existing test as an example of how a new test should be defined:

```
1 ! Vector of length = 4 -- incx = 3, n=2 -- inexact result
2     ALLOCATE (x(4))
3     testInfo%inVector=>x
4     x = [two, three, four, five]
5     nVal = 2
6     incxVal = 3
7     initStat = allFalse
8     expStat = [.FALSE.,.FALSE.,.FALSE.,.FALSE.,.TRUE.]
9     expRes = SQRT(twenty-nine)
10    testNo = testNo + 1
11    testInfo%testdescript = 'incx = 3, inexact result'
12    CALL indivtest(testInfo)
13    DEALLOCATE (x)
```

Line 1: provides a comment describing the test that follows,

Line 2: *ALLOCATE* just the space needed to contain the input vector, *x*,

Line 3: set the component *inVector* of the variable *testInfo* of type *testInfoType* to point to the vector *x*. *testInfo* contains a number of components that constitute all the required data to run and check an individual test.

Line 4: define the data in the *x* vector. This may be achieved using a block of code or, for shorter examples, an array initialization,

Line 5: set the number of values to be used. This should either be the number of elements in the input vector, *x*, i.e., *SIZE(x)* when *incxVal* is one or the number of elements that should be used if *incxVal* is greater than one; i.e., *x(j)* for  $j = 1, 1+incxVal, \dots, 1+incxVal \times (nVal-1)$ .

Line 6: set the increment to be used when accessing the *nVal* elements within the vector *x*; i.e., *x(j)* for  $j = 1, 1+incxVal, \dots, 1+incxVal \times (nVal-1)$ .

Line 7: define the initial settings for the IEEE status flags on entry to the *\*nrm2* routine. Usually these will be set to all *.FALSE*. (the *LOGICAL, PARAMETER* array *allFalse* may be used to do this) so that only the flags set to signalling by the routine are visible. This array may also be used to test that any signalling flags are not incorrectly reset when the *\*nrm2* routines are called.

The order of the flags is:

*[overflow, divide – by – zero, invalid, underflow, inexact]*

Line 8: define the expected settings of the IEEE floating point status flags on exit from the calls to each of the *\*nrm2* routines (it is assumed that the same flags will be returned for all four of the routines).

Line 9: set the expected result. This should be the result obtained when taking into account the use of floating point arithmetic.

Line 10: increment the test number, *testNo*.

Line 11: set a short descriptive title for the test. This is only output when detailed output is requested (see Section 2.3).

Line 12: makes the call to execute the individual test using *indivTest*. There is one mandatory argument, *testInfo*, which is a structure, of type *testInfoType* (see Section 2.5 for details) holding all the information required to run the test and compare the computed result against the expected.

Line 13: *DEALLOCATE* the input data vector, *x*.

### 3 The Benchmark Program

The benchmark program may be used to compare the execution speed and accuracy of the proposed hybrid method with an alternative implementation of the *\*nrm2* routines.

By default the elements of the data vectors used are random in the range (0,1). The program then times the proposed and comparison methods and calculates the relative errors in the computed results. For each value of *n*, a number of data vectors are generated and both the average execution times and the maximum relative errors are recorded. Details of the other types of data vectors that may be used along with information on how to change the extent of the benchmarks may be found in Section 3.1. A number of these input data vector options were used for testing the *FaithfulNorm* implementation described in [4]

The values of *n* used in the current release are  $10^p$ , for  $p = 1, 2, \dots, 5$  and the number of vectors generated is  $50 * 10^m$ ,  $m = 5, 4, \dots, 1$  respectively. These values may easily be altered by changing the statements as described in the declaration section of program, *sdBenchmark*, in the file *sdBenchmark.f90*.

The output generated by the program provides a synopsis of the execution times and the accuracy as well as two sets of ratios; one of the proposed hybrid method execution times over the comparison method execution times and the other, the reciprocal of the first. Values of the first set, labelled *Remark/Other*, that are greater than one indicate the factor by which the comparison method is faster than the proposed hybrid method. For the column labelled *Other/Remark*, values greater than one indicate the factor by which the hybrid method is faster than the comparison method.

#### 3.1 Optional Benchmark Data File

If a file with the name *BenchData.dat* exists in the directory where a benchmark executable is run then this file is read and the data used to alter a number of the benchmark parameters.

The format of this data file is as follows:

**Line 1:** *precision* (integer)

Choose whether to run either the single or double precision benchmark or both:

**0:** run both single and double precision benchmarks, (*Default*)

**1:** run single precision only,

**2:** run double precision only.

**Line 2:** *types* (integer) Choose whether to run either the real or complex routines or both:

**0:** run both real and complex routines, (*Default*)

**1:** run real routines only,

**2:** run complex routines only.

**Line 3:** *timeAccuracy* (integer)

Choose whether to perform either timing or accuracy measurements or both:

**0:** both time and accuracy, (*Default*)

**1:** time only,

**2:** accuracy only.

**Line 4:** *signElt*s (integer)

Sets the sign pattern to be applied to the elements of the data vector. All data vector options determined by *dataDesc* (see below) generate only positive elements. The selected sign pattern is then applied.

The currently available options are:

**0:** signs of elements are left unchanged, (*Default*)

**1:** the sign of each element is assigned randomly,

**2:** all elements are set positive,

**3:** all elements are set negative.

At the moment options 0 and 2 have exactly the same effect; they have both been made available for possible future extensions to the package.

**Line 5:** *dataDesc* (integer)

Choose the type of input vector required. If the value chosen is one of the user defined ranges (*userDefRandom*, *userDefAscending*, *userDefDescending*) then it is necessary to provide further information. See details after the list of options:

**1:** *zeroOneRandom* – random values in the range  $(0, 1)$  (*Default*).

**2:** *zeroOneAscending* – random values in the range  $(0, 1)$  sorted into ascending order. This is the worst case for the current LAPACK routines in that it forces a change of scaling factor for every element in the vector.

**3:** *zeroOneDescending* – random values in the range  $(0, 1)$  sorted into descending order.

**4:** *userDefRandom* – random values in a user defined range. Details of how to define the intervals to be used are given at the end of this section.

**5:** *userDefAscending* – as 4. above with the data sorted into ascending order.

**6:** *userDefDescending* – as 4. above with the data sorted into descending order.

**7:** *simpleAscending* – a data vector containing a simple, monotone increasing, sequence of elements, s.t.  $x_i = 1 + (i - 1)/n, i = 1, \dots, n$ . Range:  $[1, 2)$ .

- 8: *simpleDescending* – a data vector containing a simple, monotone decreasing, sequence of elements, s.t.  $x_i = 1 + (n - i)/n, i = 1, \dots, n$ . Range:  $[1, 2)$ .
- 9: *spuriousUflow* – a vector containing data that will cause harmless underflows (i.e., setting to zero will not affect the result). The vector length requested,  $n$ , must be greater than 2. Then  $x_1 = 2^{e_{max}-1}$ ,  $x_2 = 2^{e_{max}-t}$  where  $t = 24$  (single precision) and  $t = 53$  (double precision).  $\{x_i\}_{i=1}^n$  are chosen to be in the exponent range  $[eu, -1]$ .
- 10: *noOflow* – given the value of  $n$ , random elements are chosen so as to guarantee that the value of the resultant  $l_2$  norm is representable as a valid floating point number in the chosen precision. The range used is dependent on  $n$  and is almost the whole range available except that the maximum exponent used is reduced. Note: this data could still cause an overflow to occur if no scaling is applied, for example, using a simple square and add loop. Exponent range:  $[eu, e_{max} - t]$  where  $t = \lceil \log_2(n) \rceil + 2$ .
- 11: *aroundOne* – all the randomly generated values are around one in magnitude. Exponent range:  $[-5, 5]$ .
- 12: *reallySmall* – all the randomly generated values are very small in magnitude although the interval chosen does span the low and middle ranges of both Blue and Kahan’s algorithms. Exponent range:  $[eu, est]$ .
- 13: *normGradUflow* – all randomly generated values are denormalized numbers. Exponent range:  $[eu, e_{min} - t]$  where  $t = \lceil \log_2(n) \rceil + 2$ .
- 14: *fullExpRange* – the randomly generated values are chosen from the whole range of floating point numbers available. Note that it is technically possible that the resultant Euclidean norm value is not representable in the floating point format being used. To guarantee the result will be representable, see the *noOflow* option above. Exponent range:  $[eu, e_{max}]$ .
- 15: *lowHybridScale* – the randomly generated values are chosen to cause the underflow flag to signal when using the simple loop in the hybrid method. This will result in a call to Kahan’s routine, a second pass through the data and scaling. The first value in the data vector has been chosen to cause an underflow and the rest of the values all lie in the low range defined for Kahan’s method.
- 16: *highHybridScale* – the randomly generated values are chosen to cause the overflow flag to signal when using the simple loop in the hybrid method. This will result in a call to Kahan’s routine, a second pass through the data and scaling. The first value in the data vector has been chosen to cause an overflow and the rest of the values all lie in the high range defined for Kahan’s method.
- 17: *bothHybridScale* – the randomly generated values are chosen to cause both the overflow and underflow flags to signal when using the simple loop in the hybrid method. This will result in a call to Kahan’s routine, a second pass through the data and scaling. The first two value in the data vector has been chosen to cause an underflow and overflow respectively and the rest of the values all lie in the low and high ranges defined for Kahan’s method.
- 18: *KahanComp* – this is as *highHybridScale* above but the element causing the overflow is the last element rather than the first. This causes a worst case input vector for the original version of Kahan’s method.

If *dataDesc* options *userDefRandom*, *userDefAscending*, or *userDefDescending* are chosen, then more input data is required to define the data ranges to be used when generating the element values. A data range is specified by a pair of integers  $(minExp, maxExp)$  which defines the range of element values to be  $(2^{minExp}, (2 - \epsilon)2^{maxExp}]$ , i.e., any valid floating point number, except  $2^{minExp}$ , where the exponent is in the range  $[minExp, maxExp]$ . To make the data generation as flexible as possible, multiple intervals are allowed; no checks are performed on the given interval values so the defined intervals need not be disjoint.

The form of the additional data is:



**Line 6:** *numIntervals* (integer)

Provide the number of range definitions (i.e., pairs of exponent values) required.

**Line 7–6+*numIntervals*:** (pairs of integers)

Defines the minimum and maximum exponent values that are used to define each interval. The order of these values is actually immaterial.

If more than one interval is defined the number of elements generated in each interval is  $n_{int} = \lfloor n/numIntervals \rfloor$  except the last interval which contains  $n - (n - 1)n_{int}$ . The elements are generated a block at a time contiguously in the vector.

If the file, *BenchData.dat*, contained the following lines then just the routine *SNRM2* (single precision, real) would be used for accuracy testing only. The input vector would contain elements whose magnitudes were all in the range  $(1024, 16384) = (2^{10}, 2^{14})$  and whose signs were randomly chosen.

```
1 : single precision only (i.e., restrict to snrm2 and scnrm2)
1 : real routine only (i.e., restrict further to snrm2)
2 : only perform the accuracy tests
1 : set the sign of each element randomly
4 : select user defined data; random values in a specified range
1 : number of ranges to be defined
10 13 : magnitude of elements in the range (2**10, 2**14)
```

If a data file is not present then the benchmark is run with the defaults shown in the data description above; i.e., all four routines (single and double precision, real and complex) are run for both timing and accuracy on data vectors where all the elements are randomly generated in the range (0,1). This is equivalent to a *BenchData.dat* file containing

```
0 : run single and double precision routines
0 : run real and complex routines
0 : perform both timing and accuracy tests
0 : the sign of each element is left unchanged (equivalent to 2 in this case)
1 : random data in the range (0,1)
```

The subdirectory *Datafiles* of *Benchmarks* contains a number of example data files along with a *ReadMe.txt* file providing details of their settings. To use any of these files as input to the benchmark executables, it must first be renamed *BenchData.dat* in the same directory as the benchmark is being executed.

## 4 Structure of Source Files

The software bundle contains two packagings of the new hybrid code (described in the accompanying article [6]) along with a comprehensive set of test data and a benchmark code that allows comparison of the efficiency and accuracy of the new code against other implementations. Makefiles are also included to assist with building both the test and benchmark executables.

Wherever possible include files are used so that common code does not have to be duplicated to produce single and double precision versions of the routines.

The following directory structure has been used:

### User Documentation (*Doc/*)

***userManual.tex*:** the Latex sources for this user manual; this file includes *BigTab.tex*, *Introduction.tex*, *Testing.tex*, *Benchmark.tex*, *fileStructure.tex*, *Makefile.tex*, *Upate.tex* and *Results.tex*,

***userManual.pdf***: a pdf version of *userManual.tex*,  
***refs.bib***: the Bibtex data required by *userManual.tex*.

### Algorithm Sources (*Hybrid*/)

***nrm2HybridMod.f90***: a Fortran module containing single and double precision/real and complex versions of both the hybrid routines (*snrm2*, *dnrnm2*, *scnrm2*, *dznrnm2*) and stand-alone implementations of the one pass Kahan routines (*snrm2Kahan*, *dnrnm2Kahan*, *scnrm2Kahan*, *dznrnm2Kahan*).

This also includes a generic definitions for the names *nrm2* and *nrm2Kahan* which call the relevant routine depending on the type of the input vector.

***nrm2HybridSubs.f90***: a file containing the eight individual Fortran routines implementing the hybrid and one pass Kahan algorithms in single and double precision/real and complex.

This version could easily be incorporated into an existing non-module based Fortran library file.

***onePassRealKahan.inc*, *onePassComplexKahan.inc***: include files containing the common code within the single and double precision versions of the one pass version of the Kahan real and complex routines.

***realHybrid.inc*, *complexHybrid.inc***: include files containing the common code within the single and double precision versions of the hybrid real and complex routines. Used in *nrm2HybridMod.f90*, *nrm2HybridSubs.f90*, *nrm2HybridOrigKMod.f90* and *nrm2HybridOrigKSubs.f90*.

***HybridTail.inc*, *setStatusFlags.inc***: include files used by hybrid routines to prevent repeating sections of code in multiple routines. Used in *nrm2HybridMod.f90*, *nrm2HybridSubs.f90*, *nrm2HybridOrigKMod.f90* and *nrm2HybridOrigKSubs.f90*.

***nrm2HybridOrigKMod.f90***: a Fortran module containing single and double precision/real and complex versions of both the hybrid routines (*snrm2*, *dnrnm2*, *scnrm2*, *dznrnm2*) and stand-alone implementations of the original version of the Kahan routines (*snrm2Kahan*, *dnrnm2Kahan*, *scnrm2Kahan*, *dznrnm2Kahan*).

This also includes a generic definitions for the names *nrm2* and *nrm2Kahan* which call the relevant routine depending on the type of the input vector.

***nrm2HybridOrigKSubs.f90***: a file containing the eight individual Fortran routines implementing the hybrid and the original version of the Kahan algorithm in single and double precision/real and complex.

This version could easily be incorporated into an existing non-module based Fortran library file.

***realKahan.inc*, *complexKahan.inc***: include files containing the common code within the single and double precision versions of the Kahan real and complex routines. Used in both *nrm2HybridOrigKMod.f90* and *nrm2HybridOrigKSubs.f90*.

***CInterfaceNrm2.f90***: interface routines to allow the hybrid and Kahan routines to be called from a C compiler that is interoperable with the Fortran compiler being used.

*Note*: the method required for building executables when using the C interoperability features of Fortran may differ from one Fortran/C compiler combination to another. This may necessitate changes to the Makefile in the *Testing* directory in order to build the test program for these routines (*sdTestCInterface*).

***set\_precision.f90***: module used by the hybrid and Kahan module (as well as alternative *\*nrm2* implementations and driver programs) to set the kind parameters for single (*skind*) and double (*dkind*) precision. This code is portable standard Fortran.

**Testing Material (*Testing*/)** This directory contains the test module described in Section 2 of this manual plus drivers for the new hybrid and Kahan codes, and all the other BLAS implementations supplied with the package (described under *Other Versions* below).

**Makefile, runTests:** makefile and bash shell script for building and running the various testing executables; see Section 5 for more details.

**sdTest.f90:** This is the common main program used to run the full test suite (i.e., single and double precision, real and complex input data) on a version of the BLAS Level 1 *\*nrm2* routines. These routines are assumed to be contained within a module named *nrm2\_other*. The code requires the module *testSubs* to pass the four specific routines (*dnrm2*, *snrm2*, *dznrm2*, *scnrm2*) and the character variable, *descript*, which is used as a heading in the output file to identify the particular set of routines under test.

For more details on both how to control the scope of the testing and how to alter the amount of detail being output see Section 2.3.

Tests 49 to 51 require that gradual underflow is implemented on the platform being used to run the tests. The test software calls the function *ieee\_support\_underflow\_control* to check that gradual underflow is supported for single and double precision. If this is not the case for either or both precisions, a message is output on the standard output channel and the tests are skipped.

**sdTestMod.f90, sdTestMod.inc, indivTest.inc:** *sdTestMod.f90* contains two subroutines, *runDpTests* and *runSpTests* which will run the single and double precision versions of the test suite respectively. These two routines act as wrappers for the include file, *sdTestMod.inc*, which actually defines all the tests along with expected results and IEEE status flags. The include file *indivTest.inc* contains routines that run the individual real and complex versions of the tests for each precision.

The use of include files allows much of the same test code to be used for each precision.

**testSubsHybrid.f90, testSubsGen.f90:** These files all contain a module by the name of *testSubs* which imports the names as described under *sdTest.f90* above. This intermediate module is required because we may not have control over either the module name containing the BLAS routines or the names of the four routines under test. See Section 6 for details of how to test a new implementation of the *\*nrm2* routines.

**checkTests.f90:** module used to check test results and provide the user with a choice of written reports.

**sdTestBLAS.f90:** file used to link the test suite to either an existing library of BLAS routines that will include implementations of the *\*nrm2* routines or to a file containing just the routines under test (i.e., not contained within a module).

**sdTestCInterface.c:** simple driver program to test the C interface to the new hybrid routines (in *CInterfaceNrm2.f90* within the *Hybrid* directory). See also the ‘Note’ under the description of *CInterfaceNrm2.f90* above.

**sdCheckTrue.f90, checkTrueMod.f90, indivTestTrue.inc:** Driver program and equivalent files to *sdTestMod.f90* and *indivTest.f90* that may be used to check the expected results for each of the tests in the test suite that returns a finite value against the oracle value.

**Benchmark Material (*Benchmarks/*)** This directory contains the Benchmark and supporting software described in Section 3 of this user manual.

**Makefile, runBenchmarks:** makefile and bash shell script for building and running the various benchmark executables; see Section 5 for more details.

**sdBenchmark.f90:** driver program that may be used to compare the efficiency (execution time) and accuracy of the hybrid method against another implementation.

**GenerateTestVectors.f90:** module containing routines that generate test vectors in both single and double precision. (See Section 3.1 for details of how to generate different types of input data vectors when running the benchmark software.) Generic interfaces are used to allow the same calls to be made for both single and double data.

**oracle.f90:** routines that provide an extended precision computation of the  $l_2$  norm of a vector using a simple square and add loop; these are used to compute the errors in the results obtained by the various implementations. The multiple precision package (Bailey [2]), provided in the directory *MpPackage*, is used to generate the results required the double precision data. This is done because not all Fortran compilers provide quadruple precision or provide it in an inconsistent fashion.

The oracle module also checks that the value of the norm of the data vector being processed by the generic routine (*VecOracle*) or the two specific routines (*spVecOracle* – single precision data and *dpVecOracle* – double precision data) is representable in the precision being used to define the data. If the value is not representable then the module variable, *normOverflow*, is set to *.TRUE.* otherwise it is set to *.FALSE.* NOTE: this module variable is not used within the current benchmark software.

The file, *testOracle.f90*, contains a small test program that checks the accuracy of the oracle computations for a simple vector. It also checks that the oracle routine successfully detects that the input data vector would cause an overflow if computed in the requested precision.

**randMod.f90:** module containing two versions of a double precision random number generator used to create random test vectors:

1. The two routines *dpPortRand* and *seed* implement the portable random number generator from Schrage [14]. This was never meant as a good quality 64-bit generator but is both efficient and portable. The use of the *fract* variable should ensure that the lower order bits are not all zero, they will just not be very random! *seed* is used to set the initial value of the generator and must be a 32-bit integer.
2. The routines, *uni64* and *fillu*, implement the generator published by Marsaglia and Tsang [12]. This generator is a very good 64-bit generator and is included for those users who believe that the quality of the random elements may affect the accuracy of the final results!

Warning: this routine will fail for 32-bit integers if the compiler traps integer overflows.

Also included is the function *randomInt* which generates a random integer in the range  $[m, n]$  used to set the exponent range of data values.

**quicksort.f90:** set of sorting routines used to sort input data vectors into ascending or descending order. These routines are only used if the user options *zeroOneAscending*, *zeroOneDescending*, *userDefAscending* or *userDefDescending* have been chosen for *dataDesc* (see Section 3.1 for details). There is also a generic interface, *qsort*, which allows either single or double precision data to be sorted. A full description of the implementation of the quicksort routines that are used may be found in [5].

### Other Existing or Alternative Implementations (*Other Versions*/)

This directory contains other reference implementations of the *\*nrm2* routines that have been published (either as code or as detailed algorithm definitions). Also included are a number of variations of these modules which were generated during the course of this work to try and improve the performance of these original codes. For convenience these sets of routines have all been packaged as modules although the original code is largely unaltered.

**BlasOrig.f:** module containing the original Fortran 66 BLAS Level-1 routines as published in [11].

**newOrig.f90:** the original published code restructured to improve readability; this is a slightly improved version of the routine that appeared in [8]. This code uses *newOrigComplex.inc* and *newOrigReal.inc* to avoid duplication of code in the single and double precision versions.

***newOrigCsum.f90***: an extension to *newOrig.f90* which uses compensated summation to improve accuracy when collecting the sums of squares. This code uses *newOrigComplexCsum.inc* and *newOrigRealCsum.inc* to avoid duplication of code in the single and double precision versions.

***Blue.f90***: an implementation in Fortran 90 of Blue’s triple accumulator algorithm which was presented in Ratfor in [3]. The code uses *blueComplex.inc* and *blueReal.inc* to avoid duplication of code in the single and double precision versions.

***BlueCsum.f90***: an extension to *Blue.f90* which uses compensated summation to improve accuracy when collecting the sums of squares. This code uses *blueComplexCsum.inc* and *blueRealCsum.inc* to avoid duplication of code in the single and double precision versions.

***NewBlue.f90***: an implementation of Blue’s algorithm using only two accumulators. A pseudocode version of this is provided in [6]. The code uses the include files *newBlueComplex.inc* and *newBlueReal.inc* to avoid duplication of code in the single and double precision versions.

***lapack.f90***: the routines included as part of the Lapack distribution [13] (version 3.6.1 and earlier). The BLAS library routines used by the package [1] are provided in case no platform/compiler specific library is available.

***lapackCsum.f90***: an extension of *lapack.f90* which uses compensated summation to improve accuracy when collecting the sum of squares.

***lapackCsSc.f90***: an extension of *lapackCsum.f90* which attempts to improve accuracy further by using scale factors that are a power of two rather than element values in the input vector.

***origKahan.f90***: an implementation of the original version of Kahan’s algorithm alone. A pseudocode version of this is presented as Algorithm 5 in [6]. The code uses the include files *realKahan.inc* and *complexKahan.inc* from the *Hybrid* directory to avoid code duplication in the single and double precision versions.

***onePassKahan.f90***: an implementation of the one pass version of Kahan’s algorithm alone. A pseudocode version of this is presented as Algorithm 6 in [6]. The code uses the include files *onePassRealKahan.inc* and *onePassRomplexKahan.inc* from the *Hybrid* directory to avoid code duplication in the single and double precision versions.

***norm2Intrinsic.f90***: a set of wrapper routines, *\*nrm2*, that use the intrinsic function *norm2* introduced into the 2008 Fortran standard [10], as a means of computing the norm. The wrapper routines are used to provide routines of the same name as the original BLAS routines and to ensure that special cases are handled consistently.

## 5 Makefiles

Makefiles have been provided in both the *Testing* and *Benchmarks* directories. These compile and link executables for both the proposed hybrid and all the supplied alternative versions.

Both makefiles depend upon an include file, *Makefile.inc*, which is assumed to be available in the *Submission* directory. This sets the compiler to be used, the compilation and linking options and the path to an existing BLAS library (this is necessary only if the *sdTestBLAS* executable is required to run the test suite on an existing library). An example include file is provided in the distribution but this will need to be edited for the particular compiler/platform combination that the software is being installed/tested on.

### 5.1 Makefile: Include File Details

The following variables need to be set to values pertinent to the user’s system (the example settings given below refer to the NAG Fortran compiler):

**F95:** the Fortran compiler to be used.

The source code uses a number of features from the 2003 Fortran standard [9], for example, the use of square brackets in array initialization statements.

One of the implementations tested and benchmarked is the Fortran intrinsic function, *norm2*, which was introduced with the 2008 Fortran standard [10]. If this function is not available from the compiler being used, then the executables, *sdTestNorm2* and *sdBenchNorm2*, should be removed from the list of executables to be made in the *Testing* and *Benchmarks* makefiles, respectively.

The software also requires the IEEE arithmetic modules, *ieee\_arithmetic*, *ieee\_features* and *ieee\_exceptions* to be available.

Example: F95=nagfor

**F95FLAGS:** any flags to be used in the compilation phase of the Fortran compiler. These may be used to optimize the execution speed, set code warning levels, invoke tools for source code coverage, etc.

Example: F95FLAGS=-O3

**F95LINKFLAGS:** any flags that need to be passed at the linking stage of the compilation. These may be used to ensure the use of static or dynamic libraries, help with debugging, etc

Example: F95LINKFLAGS=

**CC:** a C compiler that will interoperate with the Fortran compiler defined by **F95** above. This is only required to run the executable *sdTestCInterface* (see Section 5.2). This may be left undefined if this executable is not required.

Example: CC=gcc

**CCFLAGS:** any flags that needs to be used in the compilation phase of the C compiler. Note that this variable only needs to be set if the CC variable above has been set.

Example: CCFLAGS=-O3

**LIBS:** the path to a library file containing pre-compiled versions of the BLAS Level 1 *\*nrm2* routines. Typically such libraries are provided as part of the compiler suite or via the use of a commercial numerical library.

This variable is only used to build the executables *sdTestBLASLib* and *sdBenchBLASLib* (see Sections 5.2 and 5.3). If these executables are not required then the variable may be left undefined.

Example: LIBS=

If you successfully generate results for the test and benchmark runs using a platform not given in the *Platforms.txt* file in the *Results* directory, please send a copy of the output files along with details of the compiler, operating system and hardware used to *t.r.hopkins@kent.ac.uk*. Files associated with new platforms will be added to the ‘official’ release and an appropriate acknowledgement made. It would also be interesting to hear of any library implementations that have been tested and any further test that have been added.

## 5.2 Testing Makefile

The *makefile* (in the directory *Testing/*) may be used (with *gmake*) to build a set of executables which will run the extensive test suite supplied with this package and described in Section 2. This suite aims to be as comprehensive as possible in testing the accuracy of the four routines for computing the Euclidean norm of a given input vector which were originally published as part of the BLAS Level 1 package [11].

The executables test a range of algorithms/implementations for computing the Euclidean norm. It is assumed that IEEE arithmetic is available to the software via the Fortran intrinsic modules, *ieee\_features*, *ieee\_exceptions* and *ieee\_arithmetic*. The way in which *Inf* and *NaN* values are dealt with and the setting of IEEE arithmetic status flags are described in detail in the accompanying article [6].

The following executables may be built using the file *Makefile* in the *Testing* directory:

1. *sdTestHybrid*: new hybrid version using simple loop and Kahan's algorithm and described in detail in the main article [6].
2. *sdTestOnePassKahan*: Kahan's algorithm (one pass version) on its own. This is computationally less expensive than the original version when scaling of the input data is required. A pseudo-code version is presented as Algorithm 6 in [6].
3. *sdTestOrigKahan*: Kahan's algorithm (original version) on its own. This is described in detail in the main article [6] where a pseudo-code version is also given as Algorithm 5.
4. *sdTestOrigBLAS*: original BLAS routines (Algorithm 539) with the two parameters *cutlo* and *cuthi* set to their defined values for IEEE arithmetic. This implements minor code changes from the original code which are detailed in the accompanying article [6].

It should be noted that, for organizational reasons, the original BLAS Level 1 routines have been packaged into a module, *nrm2\_Other*. This should have no effect on the results of either the testing or benchmark codes.

5. *sdTestNewOrig*: more readable/maintainable version of 4 originally described in [8] and improved further here.
6. *sdTestBlue*: Blue's algorithm is detailed in [3] where a pseudo-code version is also provided. It is also described in [6].
7. *sdTestNewBlue*: the two accumulator variant of Blue's algorithm is detailed in [6] where a pseudo-code version is also presented as Algorithm 4.
8. *sdTestLa*: version of the BLAS Level 1 routines now available (January 2017) via netlib and distributed with Lapack software (version 3.6.1, June 2016). A pseudo-code version is presented in the accompanying article [6].
9. *sdTestNorm2*: version using the new Fortran (2008+) intrinsic function, *norm2*. This uses wrapper routines to deal with special case and ensure that the resultant routines act in the same way as the originals.
10. *sdTestNewOrigCsum*: version of 5 with compensated summation implemented.
11. *sdTestBlueCsum*: version of 6 with compensated summation implemented.
12. *sdTestLaCsum*: version of 8 with compensated summation implemented.
13. *sdTestLaCsSc*: version of 8 with compensated summation and intermediate scaling factors forced to be powers of two to reduce rounding errors.
14. *sdTestCInterface*: version that calls the C interface routines for the hybrid package.
15. *sdCheckTrue*: version that uses extended precision to check the test suite to ensure that the 'true' values provided by the suite are correct.
16. *sdTestHybridSubs*: the new hybrid version using a simple loop and, if necessary, Kahan's algorithm. The subroutines are supplied as stand-alone routines in the file *nrm2HybridSubs.f90* within the *Hybrid* directory. This run should generate exactly the same results as *sdTestHybrid*.

17. *sdTestBLASSubs*: test an existing implementation of the BLAS Level 1 *\*nrm2* routines supplied as either a stand-alone set of source routines (not wrapped in a module) in one or more files or as one or more pre-compiled (.o) files.

This is a template and may require editing to successfully link.

18. *sdTestBLASLib*: test an existing platform dependent library; for example, a hardware dependent of the Level-1 BLAS specially tuned for a particular processor.

This is a template and may require editing to successfully link to the library.

It is recommended that only the computed values are tested and not the IEEE status flags as it is unlikely that precompiled versions will set these flags and this will cause a large number of warning messages to appear. This may be achieved by using

```
CALL setJustValues(.TRUE.)
```

rather than

```
CALL setJustValues(.FALSE.)
```

in two places in the file *sdTestBLAS.f90* in the *Testing* directory.

By default

```
make
```

will generate the executables 1–14. The remaining executables need to be explicitly named; for example,

```
make sdCheckTrue
```

will generate an executable to check that the test problems appear to have the correct ‘true’ solutions associated with them.

### 5.3 Benchmarks Makefile

The following executables may be built using the file *Makefile* in the *Benchmarks* directory:

1. *sdBenchOnePassKahan*: Kahan’s algorithm (one pass version) on its own. This is computationally less expensive than the original version when scaling of the input data is required. A pseudo-code version is presented as Algorithm 6 in [6].
2. *sdBenchOrigKahan*: Kahan’s algorithm (original version) on its own. This is described in detail in the main article [6] where a pseudo-code version is also given as Algorithm 5.
3. *sdBenchOrigBLAS*: original BLAS routines (Algorithm 539) with the two parameters *cutlo* and *cuthi* set to their defined values for IEEE arithmetic. This implements minor code changes from the original code which are detailed in the accompanying article [6].

It should be noted that, for organizational reasons, the original BLAS Level 1 routines have been packaged into a module, *nrm2\_Other*. This should have no effect on the results of either the testing or benchmark codes.

4. *sdBenchNewOrig*: more readable/maintainable version of 3 originally described in [8] and improved further here.
5. *sdBenchBlue*: Blue’s algorithm is detailed in [3] where a pseudo-code version is also provided. It is also described in [6].



6. *sdBenchNewBlue*: the two accumulator variant of Blue's algorithm is detailed in [6] where a pseudo-code version is also provided.
7. *sdBenchLa*: version of the BLAS Level 1 routines now available (January 2017) via netlib and distributed with Lapack software (version 3.6.1, June 2016). A pseudo-code version is presented in the accompanying article [6].
8. *sdBenchNorm2*: version using the new Fortran (2008+) intrinsic function, *norm2*. This uses wrapper routines to deal with special case and ensure that the resultant routines act in the same way as the originals.
9. *sdBenchNewOrigCsum*: version of 4 above with compensated summation implemented.
10. *sdBenchBlueCsum*: version of 5 above with compensated summation implemented.
11. *sdBenchLaCsum*: version of 7 above with compensated summation implemented.
12. *sdBenchLaCsSc*: version of 7 above with compensated summation and intermediate scaling factors forced to be powers of two to reduce rounding errors.
13. *testOracle*: an executable to check that the multiple precision package is generating plausible results for double precision data.
14. *sdBenchBLASSubs*: benchmark an existing implementation of the BLAS Level 1 nrm2 routines supplied as either a stand-alone set of source routines (not wrapped in a module) in one or more files or as one or more pre-compiled (.o) files.  
This is a template and may require editing to successfully link.
15. *sdBenchBLASLib*: benchmark an existing platform dependent library; for example, a hardware dependent of the Level-1 BLAS specially tuned for a particular processor.  
This is a template and may require editing to successfully link to the library.

By default

**make**

will generate the executables 1–12. The remaining executables need to be explicitly named; for example,

**make testOracle**

will generate an executable to check that the multiple precision package is generating plausible results for double precision data.

## 5.4 Shell Scripts

Also provided are two shell scripts, *runTests* and *runBenchmarks*, that have been set up to run the executables obtained by just using the command *make* in the *Testing* and *Benchmarks* directories, respectively.

Both scripts take a single argument which is the name of a directory to be used to store the output files generated when running the tests/benchmarks. For example,

**./runTests gfOptMac**

would, if necessary, create a directory

**../Results/Testing/gfOptMac**

and write all the resulting test output files to this directory.

If the directory already exists then the user is asked to confirm its use as this may involve the overwriting of existing files. If no argument is provided then the script exits without running any executables.

Any output generated on *stderr* during the execution of the tests/benchmarks is written to the file *RunErrorLog* in the *Testing* and *Benchmark* directories respectively.

## 6 Updating

The following subsections contain details on how to test and/or benchmark an implementation of the *\*nrm2* routines that is not available in the *OtherVersions* directory of the accompanying software package. Different approaches are required depending upon whether the new implementation is available as source code or in some non-source form, for example, a pre-compiled library or *.o* files.

Details of how to add a new test case to the existing test suite are covered in Section 2.6.1.

### 6.1 Testing a New Implementation

To test a new implementation of the *\*nrm2* routines it is necessary to link the new code with the test suite software. To facilitate the testing of the algorithms and their variants provided in the current package, the test suite software has been constructed to expect the *\*nrm2* routines to be available as a module. The test driver (*sdTest.f90*) *USES* the module *testSubs* to import the four functions *dnrm2*, *dznrm2*, *scnrm2* and *snrm2* along with a descriptive title to allow easy identification of the implementation under test (the title will appear as a header in the generated results file).

For the majority of the implementations in the current package the module containing the source files is *nrm2.Other* and the names of the routines are those given above. Listing 1 shows the required form of the module *testSubs* in this case.

Listing 1: Version used for modules in *OtherVersions*

```
1 MODULE testSubs
2 !
3 ! This module is used to pass the required routines and description
4 ! variable through to sdTest for one of the modules found in the
5 ! OtherVersions directory. All the BLAS implementations in this
6 ! directory are contained within a module named nrm2_Other and use
7 ! the normal names. The description string is set in the particular
8 ! module using the character variable descript.
9 !
10 USE nrm2_Other, ONLY: dnrm2, dznrm2, scnrm2, snrm2, descript
11 END MODULE testSubs
```

Listing 2 gives the version of *MODULE testSubs* used to test the proposed hybrid routines. Note that here the required routine names are imported from the module *nrm2HybridMod* and the variable containing the descriptive title is defined explicitly in *testSubs*.

Listing 2: Version used for the new hybrid versions

```
1 MODULE testSubs
2 !
3 ! This module is used to pass the required routines and description
4 ! variable through to sdTest from the new hybrid module.
5 ! While the routine names are as expected they are imported
6 ! from the module nrm2HybridMod rather than nrm2_other.
7 ! The description string is set here explicitly.
8 !
9 USE nrm2HybridMod, ONLY: dnrm2, dznrm2, scnrm2, snrm2
10 CHARACTER (LEN=*), PARAMETER :: descript='the new hybrid routines'
11 END MODULE testSubs
```

Finally, Listing 3 shows how the Kahan routines used in the hybrid module may be tested by using local names. As with Listing 2, the *descript* variable is set explicitly in the *testSubs* module.

Listing 3: Version used for the Kahan versions used by the new hybrid codes

```
1 MODULE testSubs
2 !
3 ! This module is used to pass the required routines and description
4 ! variable through to sdTest for the Kahan routines contained in
```

```

5  ! the new hybrid module.
6  ! The routine names are imported from the module nrm2HybridMod
7  ! rather than nrm2_other and, because they are not the expected
8  ! names, we need to change them via the USE statement.
9  ! The description string is set here explicitly.
10 !
11 USE nrm2HybridMod, ONLY: dnrm2=>dnrm2Kahan, dznrm2=>dznrm2Kahan, &
12     scnrm2=>scnrm2Kahan, snrm2=>snrm2Kahan
13 CHARACTER (LEN=*), PARAMETER :: descript = &
14     'the Kahan algorithm implementation'
15 END MODULE testSubs

```

If the new routines are already packaged as a module, then the approach used in Listing 2 may be used to interface this module with the test suite. The only changes necessary should be to use the name of the new module in place of *nrm2HybridMod* in the *USE* statement at line 9 and change the string assignment to the variable, *descript*, to something appropriate. The associated *Makefile* should then be edited to add the new module using the executable *sdTestOnePassKahan* as a template.

If the routines to be tested are available as source code but are not packaged as a module, then the simplest approach is to wrap them as a module, *nrm2\_Other*, and use Listing 1 directly. The associated *Makefile* should then be edited using the executable *sdTestOrigBLAS* as a template.

If the new implementation is not available as source code then the templates provided in the makefile in the *Testing* directory should be used to build the executables. The template *sdTestBLASLib* provides details for linking against a pre-compiled library and *sdTestBLASSubs* for routines only available as *.o* files or as source code not packaged in a module.

## 6.2 Running the Benchmark Code against New Sources

If the comparison implementation is in the form of a pre-compiled library (i.e., no source code is available) or is presented as a set of source routines not contained in a module (either in *.f/.f90/.f95* or *.o* format), then the benchmark source file *sdBenchmark.f90* needs to be changed. (An alternative for a set of routines in source format would be to wrap them in a module as described in Section 6.1 above.) The *makefile* will also need to be changed in order to build the required executable.

The following steps are required:

1. Follow the instructions in the comments at the start of the program file *sdBenchmark.f90*. The code has been set up to import the required routines, *\*nrm2*, from the module, *nrm2\_other*. Thus, if this is not the case, the routines need to be declared *EXTERNAL* and the type of each function declared explicitly.
2. To build the executables (*sdBenchBLASLib* and *sdBenchBLASSubs*) follow the comments in the file *Makefile* in the *Benchmarks* directory for these names.  
Note: the use of *sdBenchmark.f90* rather than *sdBenchmark.o*; this forces the compiler to overwrite any existing *.o* and *.mod* (or equivalent) files under all circumstances.
3. For the case where the comparison implementation is in the form of a set of routines different build definitions may be necessary depending on the ‘form’ of the routines:

**.o files:** just link

**source files:** if we have fixed format then it will probably be necessary to use a different set of compiler flags to take account of it.

If the source code is not Fortran, then an appropriate compile line will be needed as well as possible changes to the link line to accommodate the mixed language nature of the build.

## 7 Results

The directory *Results* contains the results files obtained by running the various test and benchmark programs on different compiler/hardware combinations. The file *Platforms.txt* gives details of each platform on which the tests have been run and the associated directory name within the *Results* directory.

The files *runTests* and *runBenchmarks* in the *Testing* and *Benchmarks* directories respectively may be used to write the output files directly to a user named directory. See Section 5 for more details of both the makefiles and shell scripts.

## References

- [1] 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/>
- [2] D. H. Bailey. 1995. A Fortran 90-based Multiprecision System. *ACM Trans. Math. Software* 21, 4 (Dec. 1995), 379–387. <http://doi.acm.org/10.1145/212066.212075>
- [3] 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>
- [4] 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>; retrieved November 27, 2016.
- [5] R. J. Hanson and T. R. Hopkins. 2013. *Numerical Computing with Modern Fortran*. SIAM Publications, Philadelphia, PA, USA.
- [6] R. J. Hanson and T. R. Hopkins. 2015. Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm. *ACM Trans. Math. Software* 0, 0 (2015), 00–00. Submitted for publication in ACM TOMS.
- [7] M. D. Hirschhorn. 2011. When is the sum of consecutive squares a square? *Mathematical Gazette* 95 (Nov. 2011), 511–512. <http://web.maths.unsw.edu.au/~mikeh/webpapers/paper173.pdf>
- [8] T.R. Hopkins. 1996. Restructuring Software: A Case Study. *Software — Practice and Experience* 26, 8 (Aug. 1996), 967–982.
- [9] ISO/IEC. 2004. *Information Technology – Programming Languages – Part 1, Base Language – Fortran (ISO/IEC 1539:2004)*. ISO/IEC Copyright Office, Geneva, Switzerland.
- [10] ISO/IEC. 2011. *Information Technology – Programming Languages – Fortran – Part 1: Base Language (ISO/IEC 1539:2010)*. ISO/IEC Copyright Office, Geneva, Switzerland.
- [11] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. 1979. Algorithm 539: Basic Linear Algebraic Subprograms for Fortran Usage. *ACM Trans. Math. Software* 5, 3 (Sept. 1979), 324–325.
- [12] G. Marsaglia and W. W. Tsang. 2004. The 64-bit universal RNG. *Statistics & Probability Letters* 66, 2 (Jan. 2004), 183–187. DOI:<http://dx.doi.org/10.1016/j.spl.2003.11.001>
- [13] LAPACK Project. 2015. LAPACK Official Download Site. <http://www.netlib.org/lapack/>. (2015). Accessed: 2015-06-05.

- [14] L. Schrage. 1979. A More Portable Fortran Random Number Generator. *ACM Trans. Math. Software* 5, 2 (June 1979), 132–138. DOI:<http://dx.doi.org/10.1145/355826.355828>