Proposed Consistent Exception Handling for the BLAS and LAPACK

James Demmel Jack Dongarra Mark Gates Greg Henry Xiaoye Li
Jason Riedy Weslley Pereira Julien Langou Piotr Luszczek

August 12, 2021

Abstract

Numerical exceptions, which may be caused by overflow, operations like division by 0 or sqrt(-1), or convergence failures, are unavoidable in many cases, in particular when software is used on unforeseen and difficult inputs. As more aspects of society become automated, e.g., self-driving cars, health monitors, and cyber-physical systems more generally, it is becoming increasingly important to design software that is resilient to exceptions, and that responds to them in a consistent way. Consistency is needed to allow users to build higher-level software that is also resilient and consistent (and so on recursively). In this paper we explore the design space of consistent exception handling for the widely used BLAS and LAPACK linear algebra libraries, pointing out a variety of instances of inconsistent exception handling in the current versions, and propose a new design that balances consistency, complexity, ease of use, and performance. Some compromises are needed, because there are preexisting inconsistencies that are outside our control, including in or between existing vendor BLAS implementations, different programming languages, and even compilers for the same programming language. We also propose this as a possible model for other numerical software, and welcome comments on our design choices.

1 Introduction

Sometimes it takes an event like the crash of the Ariane 5 rocket [1], a naval propulsion failure [2], or a crash in a robotic car race [3] to make people aware of the importance of handling exceptions correctly in numerical software [4]! As applications like self-driving cars, health monitors, and cyber-physical systems more generally become widespread, society's dependence on the correctness of these applications will only become more apparent. Since many of these applications are built using lower-level building blocks, often including linear algebra, it is clear that these building blocks need to be resilient to exceptions (e.g., still terminate), and respond to exceptions in a predictable, consistent way (e.g., report certain exceptions on exit) to allow higher-level applications using them to be resilient as well.

In this paper, we will explore the design space of ways to make exception handling more resilient and consistent, in particular for the widely used BLAS [5] and LAPACK [6] linear algebra libraries. While these have been widely used for decades, it turns out they do not handle exceptions consistently in a number of ways that we will describe later. We explore this design space because there is no single best solution for a number of reasons:

1. There is a cost/consistency tradeoff, with the most rigorous definitions of "consistency" potentially taking much more runtime.

2. Based on our user surveys, in which 67% of respondents said exception handling was important or very important, there is no single approach that meets all needs. These needs range from users who want the behavior of interfaces to change as little as possible (because of large amounts of existing code) to users who may want more control over the way exceptions are handled or reported. Some of these needs can be met by having different "wrappers" that offer different interfaces and semantics to the users who want them.

3. The BLAS and LAPACK are in turn based on lower-level building blocks, on whose (hopefully mostly) consistent behavior theirs depends. We will describe these building blocks, and their potential inconsistencies that we need to account for. The first building block is the IEEE 754 floating point standard, which has evolved over time. The standard recommends (but does not require) an extended precision format, implemented using 80 bits on Intel machines, and still in the x86 architecture. Inconsistencies could arise if the compiler (unpredictably) chooses to keep some intermediate results in this longer format, so our solution must accommodate this (and other) uncertainties. A correction was made in the faulty definition of min and max in the most recent 2019 version of the standard [7, 8], to assure that min and max are associative when some arguments are `NaN`s. Higher-level language standards (Fortran and C), on which BLAS and LAPACK depend, are currently modifying their specifications to include these new definitions, but it will take a while to propagate into compilers. More generally, the BLAS and LAPACK are evolving to use multiple programming languages that may define operations differently; besides min and max, this could include the product or quotient of complex scalars. Indeed, different compilers for the same language may also generate different code.

4. There are vendor-tuned BLAS implementations that may have different exceptional behavior. In some cases, this will mean that we will need to update the reference BLAS (e.g., see IxAMAX below) to assure consistency, and encourage vendors to adopt these new BLAS, including providing updated test code. In other cases that may depend on architecture-specific optimizations, it means that we will compromise on what consistency means, e.g., accepting that an exception can propagate either as an `Inf` or a `NaN`, as long as it propagates somewhere in the result.

5. Not all users may agree on the definition of "consistency". Consider an upper triangular matrix with an `Inf` or `NaN` entry above the diagonal. Are its eigenvalues well-defined, being just the diagonal entries, or not? What about a diagonal matrix with an `Inf` or `NaN` on the diagonal? Matlab currently chooses to return with a warning, and no eigenvalues reported.

6. The most rigorous definition of consistency would insist on bit-wise reproducibility, of numerical results and exceptions, from run to run of the same code on the same platform. On modern parallel architectures, where operations may be scheduled dynamically and so their order, e.g. of summation, may change from run to run, bit-wise reproducibility is not guaranteed. A simple example is summing the 4 finite numbers [x,x,-x,-x], where x+x overflows; depending on the order of summation, the result could be `+Inf`, `-Inf`, `NaN` or 0 (the correct result). Solutions for reproducible summation have been proposed, which work independently of the order of summation, but at a cost of being several times slower [9], although with opportunities for hardware acceleration [7, 8]. Intel also provides a version of their Intel(R) Math Kernel Library (Intel(R) MKL) with CNR = Conditional Numerical Reproducibility, which guarantees deterministic, and so reproducible, execution order for their multicore platforms, also with a potential performance slowdown. We leave bit-wise reproducibility to future work, but could more easily modify the current BLAS and LAPACK test codes to run each test twice, compare the results, and report any differences in a warning.

We begin in Section 2 by exploring a variety of existing exception handling inconsistencies in the current BLAS and LAPACK, as well as in underlying algorithms, programming languages and compilers, and hardware platforms, along with solutions, obstacles and/or compromises for each one. Most of the examples involve floating point exceptions, but we also give an example of integer overflow. In Section 3 we describe how to generate test code to verify that our solutions work, including using the concept of "fuzzing", which in our case means randomly inserting exceptional values in the middle of execution, even if they would not appear during a regular execution. "Fuzzing" will help test whether exceptions are propagated and handled correctly no matter where they appear in an execution (which is hard to control based just on choosing the inputs). We also discuss the challenges of using proof-based techniques to verify consistency. Finally, in Section 4 we lay out a sequence of proposed improvements to make, sorted in order of priority. Comments are welcome.

Here is some common notation that we will use later. OV is the overflow threshold in IEEE 754 arithmetic, i.e., the largest finite number (its magnitude depends of course on whether we are using single or double precision arithmetic, but the discussions below apply to either). UN is the underflow threshold, i.e., the smallest positive normalized number.

2 Exploring existing inconsistencies, obstacles, and possible solutions

We present a variety of examples of inconsistencies, possible solutions, and obstacles at various levels in the stack, from the computer arithmetic to LAPACK. We mentioned a few of these above but go into more detail here. This list is not exhaustive, and indeed rigorous testing (or proofs) are required to have confidence that (nearly) all possibilities have been found. We return to that topic in Section 3.

2.1 IEEE Arithmetic

Nearly all numerical software depends on the semantics of IEEE 754 floating point arithmetic, including the BLAS and LAPACK. (The growing use of shorter formats motivated by machine learning, e.g., bfloat16 [10], and their adoption for mixed precision linear algebra algorithms [11] are an important exception, but we leave that to future work since the BLAS and LAPACK do not yet use these formats.) In addition to changes affecting exception handling in the latest standard, the standard allows some flexibility in the semantics of operations used to evaluate arithmetic expressions, which can affect exception handling.

Section 3.7 of the 754 standard recommends, but does not require, that extended precision formats (wider than the usual single or double formats) be available, and gives lower bounds on how many more exponents and mantissa bits they should have, but does not specify exactly how many. A common implementation of this was the 80-bit format, which is still in the x86 architecture, but most compilers (not all) will generate code using the faster SSE instructions, which use the standard 64-bit format. Obviously, if a compiler decides to execute some subset of the operations in extended precision, with a larger/smaller value of OV/UN, along with additional operations to round intermediate results to and from the 64-bit format, there may be different exceptions generated. This makes reproducibility of exceptions intractable across different compilers, compiler flags, and/or architectures, but should not prevent us from still responding to exceptions that do occur in a consistent way, e.g., reporting them.

The same comments apply to other semantic possibilities, including

1. using fused-multiply-add a*b+c, where an overflow exception depends on the final value but not a*b,
2. whether underflow is detected before or after rounding,
3. the use of the default gradual underflow vs flush-to-zero, which could change whether c/(a-b) signals divide-by-zero or not, and
4. changing rounding modes, which may impact whether a final result is rounded down to the overflow threshold OV or up, causing an overflow.

One important change (a bug fix) in the 2019 standard is the added definitions of the operations min(x,y) and max(x,y), which are specified to return `NaN` if either operand is a `NaN`, so that they are associative and propagate `NaN`s. The 2008 standard did not define these, just the related operations minNum, maxNum, minNumMag, and maxNumMag, but these were defined in a way that was not associative, and might or might not propagate `NaN`s [12]. The 2019 standard changed the definitions of these related operations to make them associative too. Eventually, these new definitions will find their way into language standards (the C and Fortran standards committee are working on it) and compilers, but this will take a while, so in the meantime, we need to make sure we don't rely on them to propagate `NaN`s.

Finally, we will not depend on the 5 IEEE754 exception flags, which indicate whether an exception (invalid operation like sqrt(-1), division by 0, overflow, underflow, inexact) has occurred since the corresponding flag was the last reset. The 2008 and 2018 Fortran standards defined how to access these flags, but said that whether they were provided was processor dependent. And while they may be available on one processor, if some routines (like the BLAS) are implemented on an accelerator (like a GPU) without access to the flags, then we can't depend on them. If available, these flags can provide useful but different information than our proposed checks, and users are of course welcome to use them to see if any exception has occurred during the execution of a routine of interest. But they will not signal whether the user is passing an input containing an `Inf` or (quiet) `NaN` to a routine, or necessarily whether such a value propagates to the output or not, since operations like 3+(quiet)`NaN` (or 3+`Inf`) return `NaN` (or `Inf`) without

signaling an exception. Of course, we rely on semantics like 3+NaN =NaN to guarantee that exceptions propagate to the output, i.e., are not "lost".

For further discussion of IEEE 754 and exceptions, see [13, 14].

2.2 Programming languages and compilers

In addition to different programming languages and compilers choosing different ways to provide IEEE 754 features as described above, different programming languages and compilers may also implement basic mathematical operations in different ways, with different exception behavior. Beyond min and max as mentioned above (which may or may not currently propagate NaNs), there are also the absolute value, product, and quotient of complex numbers. When implemented using their textbook definitions, they are susceptible to over/underflow even when the final result is innocuous. The Fortran standard explicitly does not specify how to implement intrinsic arithmetic operations. However, the 2008 and 2018 Fortran standards do say that the absolute value of a complex number should be "computed without undue overflow or underflow" [15], but this should be confirmed by adding some simple tests to the LAPACK test code.

Regarding complex division, there is already an LAPACK routine {C,Z}LADIV to divide two complex numbers carefully [16, 17]. ZLADIV is used 46 times in 10 different LAPACK routines (in the SRC directory). But there are hundreds of complex divisions done in 54 routines using the built-in "/" operator. Two ways forward are (1) adding test code to the LAPACK installation package to see whether "/" behaves correctly when dividing very large or very small complex arguments, and providing a warning if it does not, and (2) converting all complex divisions to use {C,Z}LADIV, with a possible performance penalty. One could also have two versions of these routines, one using {C,Z}LADIV and one using "/", and decide which one to install based on the results of the tests.

There is also disagreement on what an "infinite" complex number is. The C99 and C11 standards [18, Annex G.3] define a complex number to be "infinite" if either component is infinite, even if one component is a NaN, and define multiplication so that infinite*(finite-nonzero or infinite) is infinite, even if the 754 rules would yield both components of the product being NaN. For example, straightforward evaluation of complex multiplication yields (Inf + 0*i)(Inf + Inf *i) = NaN + NaN *i, but the C-standard-conforming compiler yields (Inf + 0*i)(Inf + Inf *i) = Inf + Inf *i. The C standard includes a 30+ line procedure for complex multiplication [19, Annex G.5.1]. There are similar rules for complex division provided in the C standard working draft [20, Annex G.5.1] (but no procedure is provided in the C standard document). In contrast, the IEEE 754 standard and 2008 and 2018 Fortran standards say nothing about handling exceptions in complex arithmetic, or how intrinsic operations like complex multiplication and division should be implemented. This is obviously a challenge for consistent exception handling as well. To accommodate this, we propose to allow exceptions to propagate either as Infs or NaNs, since either one provides a warning to the user.

The C++ standard included complex numbers as a template class available for three floating-point types float, double, and long double. The behavior of that class for other types is unspecified but most implementations permit integral template arguments which results in compile-time errors when a floating-point complex value is combined with an integral one. For instance, std::abs applied to std::complex<T>(Inf + NaN *i) returns Inf if T is either float, double, or long double. However, the same function call returns a NaN if T is the multiprecision type mpfr::mpreal from [21]. We obtain the same pattern for (-Inf + NaN *i), (NaN + Inf *i) and (NaN - Inf *i). Moreover, if T = mpfr::mpreal, std::abs returns a NaN for any of the inputs (±Inf + Inf *i), (Inf ±Inf *i), (±Inf + 0*i), and (0 ±Inf *i), and it returns 0 for the input (0 + NaN *i). Another curious operation is the complex division. For each of the standard types, float, double and long double, the divisions (0 + 0*i)/(±Inf + NaN *i) and (0 + 0*i)/(NaN ±Inf *i) results in the complex (0 + 0*i). [a] Even the most recent C++ standard draft specifies the complex number constructor postcondition in terms of equals operator which will never be satisfied for NaN inputs [22, §26.4.4]. Neither of the standard C++ library complex template classes nor its specializations include the details of handling of exceptional floating-point values. One of the ways to address the lack of compiler support for builtin complex data types is to provide a custom implementation to be used instead of the standard <complex> header.

And if one were to use Gauss's algorithm to multiply complex numbers using 3 multiplies and 5 additions instead of 4 multiplies and 2 additions, exceptions could occur differently again. While Gauss's algorithm is unlikely to provide a speedup when multiplying complex scalarsit could when multiplying complex matrices, since the cost of

matrix multiplication is much larger than matrix addition [23].

2.3 BLAS

We consider just the standard BLAS [5], which performs a single operation, not the batched BLAS, which may perform many operations with a single call. We propose that analogous consistency requirements apply to the batched BLAS as well.

2.3.1 How to interpret alpha = 0 or beta = 0 in C = alpha*A*B + beta*C, and other BLAS

GEMM in the reference BLAS, and presumably optimized versions, interprets alpha=0 as meaning that the operation to be performed is C = beta*C. In other words, A and B are not accessed, and no `Infs` or `NaNs` that could be present in A or B are propagated. This is the expected semantics, and so it is "consistent" not to propagate `Infs` and `NaNs` in this case. Similar comments apply to beta=0, where the intended semantics are C = alpha*A*B, so that `Infs` and `NaNs` in C are not expected to be propagated. Analogous comments apply to many other BLAS routines, including various versions of matrix-matrix multiplication, matrix-vector multiplication (not all), low-rank updates, triangular solve (only TRSM, not TRSV), and AXPY. Interestingly, SCAL (x = alpha*x) does not check for alpha = 0 or 1.

The GraphBLAS [24], a specification supporting graph algorithms using sparse linear algebra (more or less), demonstrates one alternative design. The GraphBLAS does not have alpha or beta scalar parameters or the transpose parameters. Instead, the GraphBLAS includes masks and descriptors in each operation, e.g. GrB_mxm (SpGEMM). The mask controls updates on each matrix element / graph edge and does not fit into the dense BLAS well. The descriptor is an opaque object that has fields controlling how C is updated and whether A and B are transposed. C can be completely cleared before the final result is written or updated "in place." In both cases, the input entries of C are accumulated through a user-defined function. Setting that function to GrB_NULL ignores the entries. So setting the descriptor field GrB_OUTP to GrB_REPLACE and passing GrB_NULL for the accumulator is the GraphBLAS equivalent to beta = 0 in the BLAS. There is no equivalent for alpha = 0. The GraphBLAS specification spends much more text on describing the options than the dense BLAS's documentation of alpha = 0 and beta = 0 above.

2.3.2 I{S,D,C,Z}AMAX and {S,D,C,Z}NRM2

The AMAX routines take a vector as input and return the index of the (first) entry of the largest absolute value. Instead of using abs(z), the complex versions use abs(real(z)) + abs(imag(z)), because it is cheaper to compute, less susceptible to over/underflow, and adequate for many purposes, e.g., pivot selection in Gaussian elimination.

The straightforward reference implementation of the real (single precision) version

```
isamax = 1
smax = abs(A(1))
do i = 2:n
   if (abs(A(i)) > smax) then
      isamax = i
      smax = abs(A(i))
   end if
end do
```

fails to behave consistently on the following permuted inputs:

```
ISAMAX([0,NaN,2]) = 3
```

and

```
ISAMAX([NaN,0,2]) = 1
```

This is because a comparison like x > y always returns False if either argument is a `NaN`. There are various ways to define the semantics to behave in a consistent manner despite exceptional inputs. Since we want to both propagate

exceptions, and point to the "same value" independent of the order of the inputs (we explain why "same value" is in quotes below), we propose the following semantics: I{S,D}AMAX should return

1. the index of the first `NaN`, if the input contains a `NaN`, else
2. the index of the first `Inf` or `-Inf`, if the input contains an `Inf` or `-Inf`, else
3. the index of the first finite value of the largest absolute value.

"Same value" is in quotes because all `NaN`s are treated as equal, i.e., it ignores the value of their mantissa fields, which could in principle contain information useful for debugging. But since this feature is seldom used, and there is no way to prioritize one `NaN` over another, we treat them all as equal. Here is a possible implementation with the desired semantics:

```
1  smax = abs(A(1))
2  isamax = 1
3  if (isnan(smax)), return ! return index of first NaN
4  do i = 2:n
5     if (.not.(abs(A(i)) <= smax)) then
6        ! either A(i) is a NaN or abs(A(i)) > smax
7        smax = abs(A(i)), isamax = i
8        if (isnan(smax)), return
9        ! return index of first NaN
10       end if
11 end do
```

The complex versions I{C,Z}AMAX are more problematic because even in the absence of exceptional inputs, overflow can cause inconsistent outputs. For example, if z1 and z2 both have the property that abs(real(zi))+abs(imag(zi)) overflows, then they will be treated as equal (to `Inf`) even if they are not, with I{C,Z}AMAX([z1,z2]) and I{C,Z}AMAX([z2,z1]) both returning 1. See Appendix A for a "simple" (≈30 lines) correct implementation, and some performance data comparisons with the existing ICAMAX..

We note that there are analogous LAPACK routines I{C,Z}MAX1 with similar functionality (and inconsistencies) that use abs(z) instead of abs(real(z))+abs(imag(z)).

See Section 2.4.3 for an example where ISAMAX helps cause a `NaN` to fail to propagate in Gaussian elimination.

The {S,D,SC,DZ}NRM2 routines, which compute the 2-norm of a vector, have similar but less serious issues. If the input vector contains two or more `Inf`s but no `NaN`s, the routines will divide `Inf`/`Inf` and return a `NaN`. So an exceptional value does propagate, but not the expected one. Version 3.9.1 of LAPACK routines {S,D,C,Z}LASSQ have the same problem, but this is fixed in LAPACK version 3.10, based on safe scaling method in Level 1 BLAS [25].

We note that [25] proposes a different way to handle `NaN`s "consistently" in I{S,D,C,Z}AMAX, returning the index of the largest non-`NaN` entry, or 1 if all entries are `NaN`s. This is proposed for consistency with Fortran's MAXLOC and MAXVAL, and with Matlab and R, or more generally when `NaN` is interpreted as "missing data". This differs from our approach, in which we want to make sure that exceptions propagate.

2.3.3 TRSV and TRSM

The TRSV routine in the reference BLAS solves a triangular system of equations $Tx = b$ for x; T may be upper or lower triangular, and unit diagonal ($T(i,i) = 1$) or not. One may also ask TRSV to solve the transposed linear system $T^T x = b$. The reference TRSV returns $x = \begin{bmatrix}1\\0\end{bmatrix}$ when asked to solve $U * x = b$ with $U = \begin{bmatrix}1 & \text{NaN}\\0 & \text{NaN}\end{bmatrix}$ and $b = \begin{bmatrix}1\\0\end{bmatrix}$, because it checks for trailing zeros in b and does not access the corresponding columns of U (which it would multiply by zero). More generally, TRSV overwrites b with x and checks for zeros appearing anywhere in the updated x, to avoid multiplying the corresponding columns of U by zero. This means solving $Ux = b$ with $U = \begin{bmatrix}1 & \text{NaN} & 1\\0 & 1 & 1\\0 & 0 & 1\end{bmatrix}$ and

$b = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix}$ yields $x = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$. So in both cases, the NaNs in U do not propagate to the result x (neither would an Inf, which if multiplied by zero should also create a NaN). However, if L is the 2-by-2 transpose of the first U above, then calling TRSV to solve $L^T x = b$, with $b = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ returns $x = \begin{bmatrix} \text{NaN} \\ \text{NaN} \end{bmatrix}$. This is the same linear system as above, but the reference implementation does not check for zeros in this case; this is inconsistent. Again, one could imagine vendor TRSV behaving differently (in Matlab, the NaN does propagate to the solution in these examples). In these cases, if NaN were interpreted to mean "some unknown but finite number", so that 0*NaN was always 0, then not propagating the NaN would be reasonable. But if NaN meant "some unknown, and possibly infinite number", then 0*NaN should be a NaN (the default IEEE 754 behavior, which we assume), and not propagating the NaN is incorrect. Analogous comments apply to TRSM.

There are several ways to make exception handling consistent. The first and simplest way is to disallow all checking for zeros. The second way is to allow checking only for leading zeros in b when solving $Lx = b$ (or $U^T x = b$, or trailing zeros in b when solving $Ux = b$, or $L^T x = b$). The reason for the second option is that some users may expect the semantics of solving $Lx = b$ to mean solving a smaller (and possibly much cheaper) linear system when b has trailing zeros, much as they expect alpha = 0 or beta = 0 to affect the semantics of C = alpha*A*B + beta*C. In contrast, zeros appearing after the first nonzero entry in x could be the result of cancellation, and so not something the user can expect in general. But they could also be a result of the sparsity pattern of L and b, for example, if L is block diagonal, and $b(i)$ is nonzero only for indices i corresponding to a subset of the diagonal blocks. On the other hand, optimized versions of TRSM, that apply operations like GEMM to subblocks of matrices, may check for zeros in different (blocked) ways, or not at all (e.g., in MKL), for performance reasons.

We propose to take the first approach, disallowing all zero checking, because it is the simplest, and ensures consistency between TRSM and TRSV. This leaves the user the option of checking for leading or trailing zeros themselves and simply changing the size of the linear system in the call to TRSV or TRSM. To support this we will provide simple routines to return the index of the first or last nonzero entry of a 1D-array, or the first or last nonzero row of a 2D-array; the latter is most compatible with blocked optimizations of TRSM. Some versions of these already exist, for the last nonzero row or column of a 2D-array, ILA{S,D,C,Z}L{R,C}.

Analogous comments apply to TPSV and TBSV.

See Section 2.4.3 for an example where TRSV helps cause a NaN to fail to propagate in Gaussian elimination.

2.3.4 GER, SYR and related routines

The GER routine computes $A = A + \text{alpha} * x * y^T$, where A is a matrix, alpha is a scalar, and x and y are column vectors. In the reference implementation of GER, if alpha = 0, the code returns immediately, so no Infs or NaNs in x or y propagate to A. If alpha is nonzero, and if any $y(i) = 0$, the code skips multiplying $y(i)$ by alpha and x. But it does not check for zeros in x. So an Inf or NaN in $y(i)$ will propagate to all entries in column i of A, but an Inf or NaN in $x(j)$ may not propagate to all entries in row j of A. And if all $y(i) = 0$, then no Inf or NaN in $x(j)$ will propagate; this is inconsistent. Vendor GER may again be different. Our proposal for consistent exception handling would allow checking for alpha=0, but not checking for zeros inside x or y. A mathematically equivalent but possibly faster implementation would scan x for any Infs or NaNs when the first $y(i) = 0$ is encountered, to decide whether it is ok to skip multiplications by zero entries of y.

The SYR routine, for $A = A + \text{alpha} * x * x^T$ for symmetric A, has a worse inconsistency: The user needs to choose whether to update the upper or lower half of A. Since the code only checks for zero entries of x^T, this means a numerically different answer can be computed if x contains 0s and NaNs, depending on whether the upper or lower half is updated.

Searching for IF statements that compare to zero indicate that the above comments also apply to the following BLAS routines (and their complex counterparts): {S,D}SPR{,2}, {S,D}SYR{,2,K,2K}, {S,D}{TB,TP}{M,S}V, and {S,D}TR{M,S}{M,V}.

There are various other BLAS1 (DOT), BLAS2 (GEMV, GBMV, S{Y,B,P}MV) and BLAS3 (GEMM,SYMM) routines that could check for zeros, but currently do not, and should continue not to do so.

See Section 2.4.3 for an example where GER helps cause a NaN to fail to propagate in Gaussian elimination.

2.3.5 Givens rotations

Both the BLAS (xROTG, xROTGM) and LAPACK (xLARTG, {S,D}LARTGP) have routines to compute Givens rotations. LAPACK introduced its own for two reasons. First, the semantics are slightly different, eg how signs of the output are chosen, as needed by the routines that call them. Second, they are designed more carefully to avoid over/underflow [26]. The BLAS versions should be changed in analogous ways to avoid over/underflow when possible, while assuring that NaNs also propagate. The paper [25] also provides more reliable versions of Givens rotations, and claims these are better than the versions in [26], which do not pass all the tests in [25], but the version of CLARTG currently in LAPACK differs from the one tested in [25].

Here are two possible specifications for consistency for generating a Givens rotation. Let x and y be the two inputs, and c, s, r and z be the four outputs (cosine, sine, length of the vector, and a single scalar from which it is possible to reconstruct both c and s). The simplest requirement would be that if either x or y is an Inf or NaN, then at least one output must be an Inf or NaN. A more rigorous requirement for {S,D}ROTG could be the following:

1. x = +-Inf and y = finite => c = 1, s = 0, r = x, and z = 0
2. x = finite and y = +-Inf => c = 0, s = 1, r = y, and z = 1
3. x = +-Inf and y = +-Inf => c = NaN, s = NaN, r = +-Inf, and z = NaN (signs may not match)
4. either x = NaN or y = NaN => c = NaN, s =NaN, r = NaN and z = NaN

For {C,Z}ROTG, which does not compute z, we would need to distinguish the cases where the real and/or imaginary parts of x and y are finite:

1. x contains an Inf and y = finite => c = 1, s = 0, and r = x
2. x = finite and y = +-Inf + i*finite => c = 0, s = +-1 + i*0, and r = Inf
3. x = finite and y = finite +-i*Inf => c = 0, s = 0 -+i, and r = Inf
4. x = finite and y = +-Inf +-i*Inf => c = 0, s = NaN, and r = Inf
5. Both x and y contain an Inf but no NaN => c = NaN, s = NaN, r = Inf
6. Either x or y contains a NaN => c = NaN, s = NaN, r = NaN

2.3.6 Level 3 BLAS routines

Level 3 BLAS routines have classical implementations that perform $O(n^3)$ flops when all input dimensions are n. This means both that reducing the arithmetic cost by up to 25% is possible for complex inputs as discussed at the end of Section 2.2, and that methods like Strassen's algorithm exist that can reduce the arithmetic cost to $O\left(n^{log_2 7}\right)$ or less. Both methods (or their combinations) can cause different exceptions to occur, but as long as they propagate to the output as expected, they should be reported consistently by the routines that call them.

2.4 LAPACK

The BLAS and LAPACK both check integer and character input parameters (e.g., matrix dimensions) for correctness (e.g., no negative dimensions) and report errors. The BLAS do so by calling XERBLA, which by default prints an error message and halts, while LAPACK also returns an integer parameter INFO, which equals -j if the j-th input parameter is the first incorrect parameter. Positive values of INFO are used to report numerical errors, for example when attempting to solve a singular system of equations, or a failure to converge. Again, at most one error is reported.

INFO=0 indicates successful execution. Of the 2003 LAPACK routines in the LAPACK SRC directory, 1556 routines have an INFO parameter.

It is possible to imagine other ways to report errors, including using a more complicated data structure than a single integer to report more details, returning the error information as a function value instead of an argument, or signaling an exception (possible in some languages, not Fortran). For backward compatibility, we propose continuing to use a single integer parameter, whose interpretation may become more complicated.

We also note that some LAPACK interfaces (e.g., Intel's oneAPI) restrict the use of returned function values to reporting on asynchronous executions. So we could consider deprecating the few LAPACK routines that return function values (e.g., I{C,Z}MAX1) and replace them with versions using output arguments. Intel has already done this for the corresponding BLAS routines.

In this section, we explore a few examples of exception handling issues in LAPACK, beyond those we have already mentioned, and explore solutions.

2.4.1 Input Argument Checking

LAPACKE provides a C language interface to LAPACK. It currently has an option for the LAPACK driver routines to check input matrices for `NaN`s, and return with an error flag. This makes sense for the drivers, such as eigensolvers, for which the mathematical problem may be considered ill-posed (but see the earlier discussion that not everyone may agree on what "ill-posed" means, e.g., computing the eigenvalues of a diagonal matrix with an `Inf` or `NaN` on the diagonal). However, `Inf`s are also problems for these routines, for the following reason: All these drivers begin by computing the norm of the matrix, checking to see whether it is so large or small that squaring numbers of that size would over/underflow, and if needed scaling the matrix to be in a "safe" range to avoid this problem. But if the matrix contains an `Inf`, scaling would multiply each entry by x/`Inf` = 0, where x is a finite number (the target norm), resulting in a matrix containing only zeros and `NaN`s, on which computation would continue, defeating the purpose of avoiding `NaN`s. Since these drivers already compute the matrix norm, it is easy to detect whether the input contains an `Inf` or `NaN` and return with INFO indicating this. If more than one input contains `Inf`s or `NaN`s, only the first one would be reported, analogous to the previous usage of INFO.

We note that the norm routines for real matrices compute max_ij abs(A(i,j)) in a way that is guaranteed to propagate `NaN`s, by not depending on the built-in max() function, which may not propagate `NaN`s. However, in the routine for complex matrices abs(A(i,j)) can overflow even if A(i,j) is finite (same issue as I{C,Z}MAX1), so these routines need to be modified to add the option of computing max_ij max(abs(real(A(i,j))),abs(imag(A(i,j)))), which is finite if and only if no `Inf`s or `NaN`s appear.

For these driver routines that already compute norms, checking inputs for `Inf`s or `NaN`s adds a trivial O(1) additional cost. For other routines that may do only O(1) arithmetic operations per input word, adding argument checking could add a constant factor additional cost. An obvious solution is to (1) perform input argument checking automatically only when the cost is negligible compared to the overall run time, and/or (2) allow the user to optionally ask for this additional checking, either for all routines (in "debug" mode), or routine by routine, using INFO as an input parameter to request this. But this should be done in a way that is backward compatible, without requiring users to change their code if they do not want to (e.g., requiring INFO to be an input parameter). This could be done using wrappers like LAPACKE around a core implementation that assumed INFO was an input variable.

Determining which routines fall under criterion (1) would require either a (rough) operation count for each subroutine, or benchmarking. This has been explored for the BLAS: Intel's MKL_DIRECT_CALL option appears to achieve a 3× FLOPS improvement for small matrices. One of its optimizations is removing argument checks, so this provides an upper bound on the performance improvement seen in MKL. In the same range, the dynamic generation approach of LIBXSMM achieves a 5x improvement through other techniques and does not avoid dimension checks; this approach was also adopted by MKL_DIRECT_CALL_JIT [27]. So while some performance is gained by removing checks, more is possible by taking a different approach.

We choose not to distinguish `Inf`s from `NaN`s in this reporting for 3 reasons. First, `Inf`s and `NaN`s can be equally problematic, and we leave it to the user to decide how to react. Second, previous examples of inconsistencies in underlying operations (like complex division) mean that some operations could generate `Inf`s, `NaN`s, or both, so we again leave it to the user to decide how to react. Third, this lets us continue to use INFO as before, with INFO=-j

indicating that the j-th input argument is problematic, i.e., contains either an `Inf` or `NaN`.

Now we give an example involving integer overflow. Several LAPACK routines require workspace supplied by the user. The user has to provide a workspace (the WORK array) and the length of this work array (the integer LWORK). Because the integer LWORK is the dimension of an array, it is INPUT only in the interface. (Similarly to integers such as M, N, or LDA.) The minimum amount of workspace needed is given by a formula in the leading comments of the LAPACK routine, but a larger workspace often enables LAPACK to perform better, leading to the concept of "optimal" workspace. "Optimal" in this context means "optimal for better performance". (And maybe the word "optimal" is a poor choice.) In LAPACKv1 (1992) and LAPACKv2 (1994), the source code read "For optimum performance `LWORK>=N*NB`, where NB is the optimal blocksize." To know the optimal blocksize, NB, the user would call ILAENV. Starting LAPACKv3 (1999), LAPACK introduced the concept of workspace query by allowing the user to query the optimal amount of workspace needed, by passing in the integer LWORK with the value -1. When LWORK=-1, this is called a "workspace query", and the LAPACK subroutine does not perform any numerical computation and only returns the optimal workspace size in WORK(1), so an integer stored as a floating point number (rounded up slightly if necessary). Once the user knows the necessary size, the user can then pass on the necessary workspace to LAPACK by calling again LAPACK with LWORK set to the size of the provided workspace. The optimal workspace can depend on the LAPACK implementations and versions used, which is why the user must query it. Also, computing the optimal workspace can be quite long and complicated, for example for the subroutine DGESDD, it takes 273 lines of code, including 20 function calls to, in turn, query their own optimal workspace sizes. Now comes the issue of integer overflow. LWORK is an integer and so can overflow. LWORK is often the largest integer in the LAPACK interface. It can be much larger than N for example. And so LWORK is often the most integer overflow prone quantity in the interface. The integer overflow threshold depends on whether LAPACK is compiled using 32-bit or 64-bit integers. (For example, the threshold is $2^{31}-1$ for signed 32-bit integers.) As an example, the largest optimal workspace for all LAPACK subroutines is for the routines xGESDD and can be at least $4*N^2$, where N is the input matrix dimension (assuming square for simplicity). If LWORK is a signed 32-bit integer, then $4*N^2$ will overflow when $4*N^2 \geq 2^{31}$, or $N \geq 23,171$. It is therefore not feasible to correctly call SGESDD with 32-bit integers, $N \geq 23,171$, and use the optimal workspace. To inform the user of this infeasibility, during a workspace query, when WORK(1) is such that LWORK would overflow, we propose using INFO to let the user know. Even if we can return a correct WORK(1), and even if the user can successfully allocate this much memory, they will not be able to call LAPACK again with LWORK equal to the true length of WORK. To address this potential exception, we propose to test if LWORK can be set correctly, by setting INTTMP = WORK(1), testing if INTTMP .EQ. WORK(1), and if not, setting INFO to point to LWORK on exit, to indicate the problem. The test INTTMP .EQ. WORK(1) will depend (as it should) on whether INTEGER is 32-bit or 64-bit. Note for developers: When computing the optimal workspace size during a workspace query, we will also need (1) to compute the optimal workspace returned in WORK(1) carefully, to avoid integer overflow, and (2) to round it up slightly, if needed, if we compute it using floating point arithmetic.

2.4.2 Output Argument Checking

This refers to providing a warning to the user that an output argument contains an `Inf` or `NaN`.

Given one error flag INFO, the priority could be:

1. report the first input argument that contains an `Inf` or `NaN`, else
2. report the first output argument that contains an `Inf` or `NaN`.

In other words, output arguments will be checked only if no inputs already contain `Inf`s or `NaN`s. Thus, information about exceptions will "propagate" by a signal that either an input or an output contains an `Inf` or `NaN`. All the different approaches discussed for input argument checking ((1) and (2)) apply here, too.

There are other reporting possibilities, for example, encoding both the first input and first output arguments that contain an `Inf` or `NaN` into one integer. Given the rarity of exceptions (hopefully), we think that we should leave it up to the user as to how diligently they want to check the input and output arguments for `Inf` and `NaN` if they get a signal. We note that since output arguments may overwrite input arguments, we need to prioritize reporting that an input contains `Inf` or `NaN`, since the user may only be able to check output arguments after execution.

2.4.3 SGESV – incorrectly not propagating exceptions

The purpose of this example is to show how inconsistent exception handling in the BLAS and lower level LAPACK routines leads to inconsistencies in higher-level drivers. The challenge of finding all such examples motivates our proposal to report exceptional inputs and outputs.

SGESV is the LAPACK driver for solving A*x=b using Gaussian Elimination. For this example, we assume we use the original non-recursive version which calls SGETF2 internally. We give a 2x2 example that shows how the inconsistencies described before in ISAMAX, GER and TRSV interact to cause a `NaN` in input A not to propagate to output x. Let $A = \begin{bmatrix} 1 & 0 \\ \text{NaN} & 2 \end{bmatrix}$ and $b = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. First, in LU factorization ISAMAX is called on $\begin{bmatrix} 1 & \text{NaN} \end{bmatrix}$ to identify the pivot, and returns 1. Next GER is called to update the Schur complement, i.e. replace 2 by 2 – `NaN` *0. But GER notes the 0 factors, does not multiply it by the `NaN`, and leaves 2 unchanged, instead of replacing it by `NaN`. This yields the LU factorization $A = \begin{bmatrix} 1 & 0 \\ \text{NaN} & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 \\ 0 & 2 \end{bmatrix}$. The first call to TRSV solves L*y=b, correctly setting y(1) = 0, and then choosing not to multiply 0*`NaN` when updating y(2) = 1 – 0*`NaN`, leaving y(2)=1. Finally, TRSV is called again to solve U*x=y, yielding $x = \begin{bmatrix} 0 \\ 0.5 \end{bmatrix}$, with no `NaN` appearing in the final output.

If one calls the recursive of SGESV introduced in release 3.6.0, then 2 – 0*`NaN` will be computed by a call to SGEMM, which may be more likely to compute a `NaN`.

The many other one-sided factorization routines should be examined to see if they are susceptible to similar problems, and fixes.

2.4.4 SSTEMR – correctly not propagating exceptions

The purpose of this example is to show that some algorithms are designed with the expectation that there will be exceptions, and handles them internally. In such cases, there is no reason to report them. Such examples are uncommon and need to be carefully documented.

SSTEMR computes selected eigenvalues, and optionally eigenvectors, of a symmetric tridiagonal matrix T. One internal operation is counting the number of eigenvalues of T that are $< s$, for various values of s. Letting D be the array of diagonal entries of T, and E be the array of offdiagonal entries, the inner loop (which appears in SLANEG) that does the counting looks roughly like this (a careful analysis is provided elsewhere [28]):

1. DPLUS = D(i) + T
2. IF (DPLUS < 0) COUNT = COUNT + 1
3. T = (T/DPLUS) * LLD(i) – s

As described in the symmetric tridiagonal eigensolvers report [28], it is possible for a tiny DPLUS to cause T to overflow to `Inf`, which makes the next DPLUS equal to `Inf`, which makes the next T = `Inf`/`Inf` = `NaN`, which then continues to propagate. Checking for this rare event in the inner loop would be expensive, so SLANEG only checks for T being a `NaN` every 128 iterations, yielding significant speedups in the most common cases. The value 128 is a tuning parameter.

Similar examples are also discussed elsewhere [29, 30], including other significant speedups.

2.4.5 Reporting internal exceptions

In addition to the reporting possibilities discussed above for input and output arguments, it is in principle possible for `Inf`s and `NaN`s to be created internally, but not propagate to an output variable. When these are not anticipated exceptions as in Section 2.4.4, then in principle the user should be made aware of these also, using the INFO parameter. This is particularly true if an internal exception should cause the subroutine to exit prematurely. This is something for which IEEE exception flags could potentially be used, but as mentioned in Section 2.1, we cannot count on them being available on all platforms. So we present here a partial solution, in the sense that it would catch only some,

but not necessarily all, "internal" Infs and NaNs, but would be quite cheap assuming we also implement input and output parameter checking. The idea is that any LAPACK routine with an INFO parameter that calls other LAPACK routines with INFO parameters will check to see if those "internal" INFO parameters ever signal that an input or output parameter contains an Inf or NaN, or there was an internal exception. When the calling routine returns, we propose to set its value of INFO as follows:

1. If the calling routine runs to completion:
   (a) report the first input argument that contains an Inf or NaN, else
   (b) report the first output argument that contains an Inf or NaN, else
   (c) report an internal exception,
2. else, if the calling routine decides to return prematurely, we report an internal exception.

Note that this definition applies recursively, through the depth of the call tree.

As in Section 2.4.2, there are a variety of ways to "report" an internal exception, for example by assigning a unique index to each internal subroutine call with an INFO parameter in the source code, and again reporting the first (lowest index) place an internal exception occurred. More complicated and informative encodings are imaginable, but we recommend this simple approach.

3 How to test consistency

Ideally, an implementation of an algorithm would come with a proof that it handles exceptions consistently, e.g., terminates, given various assumptions about underlying building blocks. This sounds like solving the halting problem, but it should be much easier in practice for the algorithms we are talking about, because of their structure: loops with finite loop bounds, or iterations that can be confirmed to include a condition that the number of iterations cannot exceed a certain finite limit (i.e., do not simply iterate until a condition like Error_Estimate $<$ Threshold holds, which will be false if Error_Estimate is a NaN). The structure of (most) LAPACK algorithms (and certainly the BLAS) is simple enough to either do this by hand or perhaps automate it, at least enough to identify those routines that require a human expert to analyze them. We leave this for future work.

In the meantime, we propose expanding our LAPACK test code to test both termination and error reporting. It is straightforward to insert Infs and NaNs into random locations in the inputs of a routine and see what happens, and this should be our first step. But this will not test what happens if an Inf or NaN is generated in an unpredictable location in the middle of an execution, which is certainly needed to test all the ways exceptions could be reported as discussed in Sections 2.4.2 and 2.4.5. To deal with this, we can use "fuzzing" [31], which in our context means randomly introducing an Inf or a NaN into one or more (randomly) chosen variables during the course of an execution. We propose fuzzing because it is very difficult to devise an input without Infs or NaNs that will, or should, generate an Inf or NaN during some intermediate calculation. Since our exception handling should be impervious to when and where exceptions are generated, fuzzing is a suitable approach. In addition to choosing random locations to introduce Infs and NaNs, we should make sure to introduce them into subroutines called by the routine being tested (and so on recursively) to make sure that our reporting mechanisms work (or identify what needs to be fixed). Fuzzing requires instrumentation tools that can follow and modify an execution of an algorithm, and confirm that all or most code segments have been executed by at least one test input. This will slow down execution significantly, but small input matrices are likely to be enough to get close to 100% coverage.

4 Proposed tasks to improve consistency, in priority order

We propose a list of tasks to improve the exception handling consistency of the BLAS and LAPACK. They are sorted in priority order, with "easy" tasks with potentially larger impact first. Comments are welcome.

1. Modify the LAPACK test code to see whether complex division (including real/complex) and complex absolute value are implemented in a way that avoids over/underflow, and issue a warning if they are not, as described in Section 2.2. Even though we are trying to avoid dependence on min/max, we should test these as well, as described in Section 2.1. Finally, we could test complex multiplication as described in Section 2.2, and simply report whether the semantics is based on the textbook definition, or the C standard (this would just be informative, not an error check).

2. Modify LAPACKE to provide the option to return with an error flag if inputs of selected driver routines contain either `NaN`s or `Inf`s. The current version only checks for `NaN`s, but as described in Section 2.4.1 this is inadequate.

3. Modify the LAPACK routines that compute complex matrix norms (e.g., {C,Z}LANGE) to provide a norm that correctly signals whether or not the input matrix contains an `Inf` or `NaN`, as in Section 2.4.1. For the driver routines that already start by computing norms of the input matrix or matrices, modify them to use INFO to report whether an input contains an `Inf` or `NaN`, and return immediately if the mathematical problem is not "well-posed," consistently with LAPACKE. Correspondingly change the LAPACK test code to confirm they behave as desired.

4. Fix the reference BLAS routines as described in Section 2.3. Update the BLAS test code accordingly to make sure `Inf`s and `NaN`s are propagated as expected. This includes devising finite inputs that create `Inf`s and `NaN`s internally, which should be straightforward given the relative simplicity of the operations performed. Encourage vendors to adopt the new BLAS. Provide a simple routine to return the index of the first or last nonzero entry of a 1D-array, or the first or last nonzero row of a 2D-array, to help users accelerate xTRS{V,M} also as described in Section 2.3.

5. Determine the best way of providing wrappers to LAPACK that allow users to choose between the following 3 functionalities:

   (a) INFO behaves as usual (including the changes in the third step above)

   (b) INFO will be used to do input and output argument checking as described in Sections 2.4.1 and 2.4.2.

   (c) INFO will also be used to report internal exceptions, as described in Section 2.4.5.

   In all the above cases, INFO would continue to be an output-only parameter. All these options would be implemented by the wrapper calling a new implementation of all LAPACK routines with INFO parameters, where INFO would be treated as an input parameter, with input values 0, 1, and 2 indicating which of the above 3 cases to choose. These routines could have names like sgesv_check to distinguish them from the existing implementation. This would also allow users to instrument subsets of their code they are interested in debugging or making more resilient.

   Since there are many possible sources of error, and one INFO parameter, we propose prioritizing what to report as follows (INFO=0 continues to mean no error):

   (a) INFO = -k if input argument k is "wrong", e.g., a negative dimension; choose the smallest k and return immediately (current practice)

   (b) INFO = -k if input argument k contains an `Inf` or `NaN` and this means that the algorithm needs to return immediately (e.g., an eigensolver) (new option)

   (c) INFO > 0 if there is a numerical failure (e.g., zero pivot) (current practice)

   (d) INFO = -k if input argument k contains an `Inf` or `NaN` and the algorithm continued to execute (new option)

   (e) INFO = some unique positive value (depends on case 3 above), pointing to the first output argument that contains an `Inf` or a `NaN` (new option)

(f) INFO = some unique positive value (depends on above cases) to point to the first place an internal exception occurred (new option)

These are numbered so that a nonzero value of INFO has the same meaning independent of which of the 3 above functionalities the user chooses. We illustrate this by using SGESV as an example. Note that we let SGESV continue executing, even if the input contains an `Inf` or `NaN`, as is the practice in Matlab.

SGESV(N, NRHS, A, LDA, IPIV, B, LDB, INFO)

(a) INFO = 0 means no error (current practice)

(b) INFO = -1 if N $<$ 0 (current practice)

(c) INFO = -2 if NRHS $<$ 0 (and INFO not already set, current practice)

(d) INFO = -4 if LDA $<$ max(1,N) (ditto)

(e) INFO = -7 if LDB $<$ max(1,N) (ditto)

(f) INFO = k, if k is first zero pivot, indicating a singular matrix (ditto)

(g) INFO = -3 if A contains `NaN` or `Inf` on input (and INFO not already set, new option)

(h) INFO = -6 if B contains `NaN` or `Inf` on input (ditto)

(i) INFO = N+3 if A contains `NaN` or `Inf` on output (ditto)

(j) INFO = N+6 if B contains `NaN` or `Inf` on output (ditto)

(k) INFO = N+9 if SGETRF reports a `NaN` or `Inf` (ditto)

(l) INFO = N+10 if SGETRS reports a `NaN` or `Inf` (ditto)

6. Update the LAPACK test code to test the features in step 5 as they are introduced. Use "fuzzing" in the LAPACK testcode as described in Section 3.

A More details on I{S,D,C,Z}AMAX

The following is a consistent implementation of I{C,Z}AMAX, according to Section 2.3.2. It is written so that its correctness should be apparent, but one can imagine various ways to optimize its performance, depending on the architecture and length n of the input. For example, the algorithm below does just one pass over the data. Also, checking for exceptions only periodically, as in Section 2.4.4, would likely be faster.

```
1  noinfyet = 1 ! no Inf has been found yet
2  scaledsmax = 0
3  ! indicates whether A(i) finite but
4  ! abs(real(A(i))) + abs(imag(A(i))) = Inf
5  smax = -1
6  do i = 1:n
7     if (isnan(real(A(i))) .or. isnan(imag(A(i)))) then
8        ! return when first NaN found
9        icamax = i, return
10    elseif (noinfyet == 1) ! no Inf found yet
11       if (isinf(real(A(i))) .or. isinf(imag(A(i)))) then
12          ! record location of first Inf,
13          ! keep looking for first NaN
14          icamax = i, noinfyet = 0
15       else ! still no Inf found yet
16          if (scaledsmax == 0)
17             ! no abs(real(A(i))) + abs(imag(A(i))) = Inf yet
18             x = abs(real(A(i))) + abs(imag(A(i)))
19             if (isinf(x))
20                scaledsmax = 1
21                smax = (.25*abs(real(A(i)))) + (.25*abs(imag(A(i))))
22                icamax = i
23             elseif (x > smax) ! everything finite so far
24                smax = x
25                icamax = i
26             endif
27          else ! scaledsmax = 1
28             x = (.25*abs(real(A(i)))) + (.25*abs(imag(A(i))))
29             if (x > smax)
30                smax = x
31                icamax = i
32             endif
33          endif
34       endif
35    endif
36 enddo
```

We also did timings for ICAMAX on an Intel(R) Xeon(R) Platinum 8180 CPU @ 2.50GHz ("2S Skylake"), 1 core/thread out of 28/56, with 38.5MB L2 and 12x16GB DDR4-2666 memory, comparing the old reference BLAS code [5] vs. the new code above vs. the latest Intel(R) MKL. Not surprisingly, Intel(R) MKL's performance in the presence of no NaNs was fastest. The reference code vs. the above new code was compiled with the Intel(R) Fortran Intel(R) 64 Compiler for applications running on Intel(R) 64, Version 19.0.8, compiled with "-O3 -fprotect-parens -mp1 -mieee-fp". Performance in the presence of a `NaN` being inserted depended on where the `NaN` was inserted. It appears that Intel(R) MKL does not do a `NaN` check first or while scanning the data. Instead, the performance suggests it does the operation and then checks if a `NaN` was present, and then calculates the position of the first `NaN` (that is, Intel(R) MKL's ICAMAX is already conformant to this new standard So this new code could be made to look arbitrarily faster than Intel(R) MKL just by inserting a `NaN` into the beginning. The new code would exit immediately regardless of how large the problem size N is, whereas Intel(R) MKL would look at the entire array. But

if the NaN were inserted at the end, then the extra performance of the vendor library would dominate.

B Test Cases

This is a draft description of how to generate BLAS test cases of length n (possible values for n: 1, 2, 3, 10, 128, ...).

The ACM Algorithm 978 for safe scaling [25] says it provides a better set of test cases for the BLAS1, but it may not address all the new exception handling semantics, e.g., for ICAMAX below. It may also be a good starting point.

ISAMAX test cases: Default entries: A(k) = (-1)^k*k

1. At least 1 NaN, no Inf s (<=15 cases)
   (a) 1 NaN, at location:
      i. 1; 2; n/2; n
   (b) 2 NaNs (if possible, i.e. n>1, ditto later)
      i. 1,2; 1,n/2; 1,n; 2,n/2; 2,n; n/2,n
   (c) 3 NaNs
      i. 1,2,n/2; 1,2,n; 1,n/2,n; 2,n/2,n
   (d) All Nans
2. At least 1 NaN and at least 1 Inf
   (a) For each example above (<=7*15 cases):
      i. Insert Inf in first non-NaN location
      ii. Insert -Inf in first non-NaN location
      iii. Ditto for last non-NaN location
      iv. Ditto for first and last non-NaN locations
      v. Insert (-1)^k*Inf in all non-NaN locations
3. No NaNs, at least 1 Inf (15 cases)
   (a) Same pattern as (1) above, inserting (-1)^k*Inf into A(k)

Total #cases < 5*(15 + 7*15 + 15) = 615

ICAMAX test cases: In addition to the above, we need additional cases because A(k) can be finite but abs(real(A(k)))+abs(imag(A(k))) can overflow.

4 cases:

1. A(k) = -k + i*k for k even, A(k) = OV*((k+2)/(k+3)) + i*OV*((k+2)/(k+3)) for k odd
   (a) (Correct answer = last odd k)
2. Swap odd and even. (Correct answer = last even k)
3. A(k) = -k + i*k for k even, A(k) = OV*((n-k+2)/(n-k+3)) + i*OV*((n-k+2)/(n-k+3)) for k odd
   (a) (Correct answer = 1)
4. Swap odd and even (Correct answer = 2)

For each of the above 4 cases, insert NaNs and/or Infs in same way as for ISAMAX.

xNRM2 test cases:

Some of the following tests use the Blue's constants b and B for the sum of squares from [32]. Any floating-point number x satisfying $b \leq x \leq B$ has its square (x^2) guaranteed not over- nor underflow.

1. Finite input which expects a correct finite output.
   (a) A(k) = (-1)^k*b/2, where b is the Blue's min constant. A(k)^2 underflows but the norm is (b/2)*sqrt(n).
   (b) A(k) = (-1)^k*x, where x is the underflow threshold. A(k)^2 underflows but the norm is x*sqrt(n).
   (c) A(k) = (-1)^k*x, where x is the smallest subnormal number. A(k)^2 underflows but the norm is x*sqrt(n). * Mind that not all platforms might implement subnormal numbers.
   (d) A(k) = (-1)^k*2*B/n, where B is the Blue's max constant, n greater than 1. A(k)^2 and the norm are finite but sum_k A(k)^2 underflows.
   (e) A(k) = (-1)^k*2*B, where B is the Blue's max constant. A(k)^2 overflows but the norm is (2*B)*sqrt(n).
   (f) A(k) = b for k even, and A(k) = -7*b for k odd, where b is the Blue's min constant. The norm is 5*b*sqrt(n).
   (g) A(k) = B for k even, and A(k) = -7*B for k odd, where B is the Blue's max constant. The norm is 5*B*sqrt(n).
   (h) A(k) = (-1)^k*2*OV/sqrt(n), n greater than 1. A(k) is finite but the norm overflows.
2. Finite input which expects a correct infinite output.
3. Input contains 1 or more `Inf`s, but no `NaN`s, so correct output is `Inf`. Test cases add 1 or more `Inf`s to all the cases under (1) and (2), in addition to adding `Inf`s to the "harmless" example: A(k) = (-1)^k*k. `Inf` locations:
   (a) 1 `Inf` at: 1; 2; n/16; n/2; n.
   (b) 2 `Inf`s at: 1,2; 1,n/16; 1,n/2; 1,n; 2,n/16; 2,n/2; 2,n; n/16,n/2; n/16,n; n/2,n.
   (c) 3 `Inf`s at: 1,2,n/16; 1,2,n/2; 1,2,n; 1,n/16,n/2; 1,n/16,n; 1,n/2,n; 2,n/16,n/2; 2,n/16,n; 2,n/16,n; n/16,n/2,n.
   (d) All `Inf`s.
4. Input contains 1 or more `NaN`s, so correct output is a `NaN`. Test cases add 1 or more `NaN`s to all the cases under (1) and (2), in addition to adding `NaN`s to the "harmless" example: A(k) = (-1)^k*k. `NaN` locations: Same as the `Inf` locations in (3). For each `NaN` location scenario:
   (a) Input contains no `Inf`s.
   (b) Input contains `Inf`s:
      i. Insert `Inf` in first non-`NaN` location;
      ii. Insert -Inf in first non-`NaN` location;
      iii. Ditto for last non-`NaN` location;
      iv. Ditto for first and last non-`NaN` locations.
      v. A(k) = (-1)^k*`Inf`.
   (c) All `NaN`s.

References

[1] Douglas N. Arnold. The Explosion of the Ariane 5. Some disasters attributable to bad numerical computing, 2000. 23 August 2000, www-users.math.umn.edu/~arnold/disasters/ariane.html.

[2] Gregory Slabodkin. Software glitches leave Navy Smart Ship dead in the water, 1998. GCN, 13 July 1998, https://gcn.com/Articles/1998/07/13/Software-glitches-leave-Navy-Smart-Ship-dead-in-the-water.aspx. Accessed 28 April 2021.

[3] [OT Roborace] Driverless racecar drives straight into a wall, 2020. https://www.reddit.com/r/formula1/comments/jk9jrg/ot_roborace_driverless_racecar_drives_straight/gai295l/.

[4] J. Demmel and E. J. Riedy. A new IEEE 754 standard for floating-point arithmetic in an ever-changing world. *SIAM News*, July/August 2021.

[5] http://www.netlib.org/blas/.

[6] http://www.netlib.org/lapack/.

[7] 754-2019 IEEE Standard for Floating Point Arithmetic, 2019.

[8] D. G. Hough. The IEEE Standard 754: One for the History Books. *Computer*, 52(12):109–112, December 2019. DOI: 10.1109/MC.2019.2926614.

[9] P. Ahrens, J. Demmel, and H. D. Nguyen. Efficient Reproducible Floating Point Summation and BLAS. *ACM Trans. Math. Software*, 46(3), 2020.

[10] https://en.wikipedia.org/wiki/bfloat16_floating-point_format, 2017.

[11] Ahmad Abdelfattah, Hartwig Anzt, Erik G Boman, Erin Carson, Terry Cojean, Jack Dongarra, Alyson Fox, Mark Gates, Nicholas J Higham, Xiaoye S Li, Jennifer Loe, Piotr Luszczek, Srikara Pranesh, Siva Rajamanickam, Tobias Ribizel, Barry F Smith, Kasia Swirydowicz, Stephen Thomas, Stanimire Tomov, Yaohung M Tsai, and Ulrike Meier Yang. A survey of numerical linear algebra methods utilizing mixed-precision arithmetic. *The International Journal of High Performance Computing Applications*, 2021. DOI: 10.1177/10943420211003313.

[12] grouper.ieee.org/groups/msc/ANSI_IEEE-Std-754-2019/background/minNum_maxNum_Removal_Demotion_v3.pdf, 2019.

[13] https://grouper.ieee.org/groups/1788/email/pdfmPSi1DgZZf.pdf.

[14] J. Hauser. Handling floating point exceptions in numeric programs. *ACM Trans. Programming Languages and Systems*, 18(2), 1996. https://people.eecs.berkeley.edu/~fateman/264/papers/hauser.pdf.

[15] 2008 Fortran Standard, 2008. https://wg5-fortran.org/f2008.html.

[16] D. Priest. Efficient scaling for complex division. *ACM Trans. Math. Software*, 30(4), 2004.

[17] M. Baudin and R. Smith. A robust complex division in SciLab, 2012. https://arxiv.org/abs/1210.4539.

[18] C11 standard, 2011. https://en.wikipedia.org/wiki/C11_(C_standard_revision).

[19] Programming languages – C, 2011. International Standard.

[20] Programming languages – C, ISO/IEC 9899:202x (E), February 5 2011. N2478 working draft.

[21] https://github.com/advanpix/mpreal.

[22] Working draft, standard for programming language C++, November 27 2019. Document number: N4842.

[23] D. S. Scott. Out of core dense solvers on Intel parallel supercomputer, 1992.

[24] Aydın Buluç, Timothy Mattson, Scott McMillan, José Moreira, and Carl Yang. The GraphBLAS C API specification, September 25, 2019. Version 1.3.0.

[25] E. Anderson. Algorithm 978: Safe scaling in the Level 1 BLAS. *ACM TOMS*, 44(1), 2017.

[26] D. Bindel, J. Demmel, W. Kahan, and O. Marques. On computing Givens rotations reliably and efficiently. *ACM Trans. Math. Soft.*, 28(2), 2002. LAPACK Working Note 148.

[27] https://software.intel.com/content/www/us/en/develop/articles/intel-math-kernel-library-improved-small-matrix-performance-using
html, 2020.

[28] O. Marques, E. J. Riedy, and C. Vömel. Benefits of IEEE-754 features in modern symmetric tridiagonal eigensolvers, 2005. LAPACK Working Note 172.

[29] J. Demmel and X. Li. Faster numerical algorithms via exception handling. *IEEE Trans. Comput.*, 43(8), 1994.

[30] J. Demmel and V. Volkov. Using GPUs to accelerate the bisection algorithm for finding eigenvalues of symmetric tridiagonal matrices. Technical Report UCB/EECS-2007-179, UC Berkeley, 2007.

[31] https://en.wikipedia.org/wiki/Fuzzing, 2021.

[32] James L. Blue. A Portable Fortran Program to Find the Euclidean Norm of a Vector. *ACM Trans. Math. Software*, 4(1):15–23, 1978.

C Footnotes and comments

[a]Behavior of complex abs and complex division. Tests on Ubuntu 18.04.5 LTS (5.4.0-70-generic) with GNU compilers, GCC version 7.5.0.