2022 IEEE/ACM Sixth International Workshop on Software Correctness for HPC Applications (Correctness) | 978-1-6654-6335-5/22/$31.00 ©2022 IEEE | DOI: 10.1109/CORRECTNESS56720.2022.00006

Proposed Consistent Exception Handling for the BLAS and LAPACK

James Demmel*, Jack Dongarra†, Mark Gates†, Greg Henry‡, Julien Langou§
Xiaoye Li¶, Piotr Luszczek†, Weslley Pereira§, Jason Riedy‖, Cindy Rubio-González**
*EECS and Math Depts., UC Berkeley, demmel@berkeley.edu
†ICL, U. of Tennessee, Knoxville, {dongarra, mgates3, luszczek}@icl.utk.edu
‡Intel Corp., greg.henry@intel.com
§Dept. of Math. and Stat. Sci., U. Colorado Denver, {Julien.Langou, weslley.pereira}@ucdenver.edu
¶AMCR Division, Lawrence Berkeley National Lab, XSLi@lbl.gov
‖Lucata Corp., jason@acm.org
**CS Dept., UC Davis, crubio@ucdavis.edu

***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. And user requests from our surveys are quite diverse. We also propose our design as a possible model for other numerical software, and welcome comments on our design choices.**

I. 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], [5]! 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, such as linear algebra, these building blocks clearly 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 to be resilient too.

In this paper, we explore the design space of ways to make exception handling more resilient and consistent, in particular for the widely used BLAS [6] and LAPACK [7] 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) 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 for code compatibility to users who want more fine-grained control over the way exceptions are handled or reported. And 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? Matlab currently chooses to return with a warning, and no eigenvalues are reported. Roughly speaking, user wishes for consistency fall into 4 (related) categories: guaranteed termination, correct mathematical behavior (*e.g.*, what an eigenvalue means, as above), propagation (*e.g.*, a `NaN` that is input to or created during a subroutine call should propagate to the output unless there is a mathematical reason that it should not), and reporting (*e.g.*, using LAPACK's INFO parameter, or new mechanisms, to flag exceptions). We present these user requests in more detail in

978-1-6654-6335-5/22/$31.00 ©2022 IEEE
DOI 10.1109/Correctness56720.2022.00006

Section III-A. Some of these needs can be met by having different "wrappers" that offer different interfaces and semantics to the users who want them, some by new subroutine arguments, and some by new routines that let users choose options for subsequent LAPACK calls.

2) The BLAS and LAPACK are based on lower-level building blocks, on whose behavior theirs depend. We describe these building blocks, and their potential inconsistencies that we need to accommodate. These include the IEEE 754 floating-point standard, higher level language standards (*e.g.*, Fortran and C), and different compilers for the same language.
3) There are vendor BLAS implementations that may have different exceptional behavior. In some cases, this means 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, for example, an exception can propagate either as an `Inf` or a `NaN`, as long as it propagates somewhere in the result.
4) There is a cost/consistency tradeoff, with the most rigorous definitions of "consistency" potentially taking much more runtime. For example, the most rigorous definition of consistency would insist on bit-wise reproducibility, of both 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.*, summation order, 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). Different orders may also produce differently signed zeros. Solutions for reproducible summation have been proposed, which work independently of the order of summation, but at a cost of being several times slower [8], although with opportunities for hardware acceleration [9], [10]. 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 building on *consistent* handling of exceptional cases.

Section II explores a variety of existing exception handling inconsistencies and possible solutions. Section II-A considers IEEE arithmetic and its differing implementations, programming languages, and compilers. We have no control over these inconsistencies, and must accommodate them. Section II-B considers the BLAS and LAPACK, giving examples of current inconsistencies, and some proposed solutions for the BLAS. Section III presents our proposed exception handling interface for LAPACK, including a list of all the user requests that it is based on, and a summary of the options provided to the user, with more details for the interface of SGESV. Section IV describes how to generate test code to verify that our solutions work, using the concept of "fuzzing", *i.e.*, inserting `Inf`s and `NaN`s at selected locations in the middle of execution. More details on all these topics are available in [11].

II. 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 IV.

A. IEEE Arithmetic, Programming Languages, and Compilers

Nearly all numerical software depends on the semantics of IEEE 754 floating-point arithmetic, including the BLAS and LAPACK. (Dealing with new, shorter formats like bfloat16 [12] is future work.) In addition to changes affecting exception handling in the latest standard [9], the standard allows some flexibility in the semantics of operations used to evaluate arithmetic expressions, which can affect exception handling. Examples include (1) the optional use of extended precision formats (see section 3.7 of the 754 standard), which may have more exponent bits, (2) using fused-multiply-add $a \times b + c$, where an overflow exception depends on the final value but not $a \times b$, (3) whether underflow is detected before or after rounding, (4) 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 (5) 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 are 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 NaNs. The 2008 standard did not define these, just the related operations minNum, maxNum, minNumMag, and maxNumMag, but because of an oversight regarding different sections' impacts these were defined in a way that was not associative, and might or might not propagate NaNs [13] when signalling NaNs are involved. The 2019 standard adapted to this oversight to ensure the operations are associative. Eventually, these new definitions will find their way into languages (the C and Fortran standard committees are working on it) and compilers, but this will take a while, so in the meantime, we need to cope with the ambiguity.

Finally, we will *not* depend on the five IEEE 754 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 implementation and hardware dependent. Similarly, the C99 standard [14] provides for a floating-point environment within which exceptions can be examined, but again its availability depends on the implementation. And while they may be available on one processor, if some routines (like the BLAS) are implemented on an accelerator (like a GPU or remote memory system) without or with only aggregate access to the flags, then we cannot depend on them.

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 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]. The C99 and C11 standards [16, Annex G.3] define when a complex number is "infinite" in more detail, with a 30+ line procedure for complex multiplication [17, Annex G.5.1]. While compilers will adopt this as a default, the compilers also provide options for using other definitions (*e.g.*, gcc's -fcx-limited-range[1]). There are similar rules for complex division provided in the C standard working draft [18, Annex G.5.1] although no procedure is provided in the C standard document. In contrast, the IEEE 754 standard and both 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.

The takeaway from these and other examples [11] is that our definition of "consistent" exception handling needs to take these unavoidable inconsistencies into account, since they are beyond our control. So our goals will be to propagate exceptions whenever they occur (unless the code anticipates and handles them appropriately), and provide appropriate run-time and install-time warnings to the user, including modifying the LAPACK test code to warn the user about how operations like complex division, etc. might cause exceptions.

B. BLAS and LAPACK

We give a few examples where floating-point exceptions are handled inconsistently by the BLAS and LAPACK and provide some suggestions for consistent handling. We also show examples where exceptions are dealt with carefully. For more examples, see [11].

First consider IxAMAX, where x=S, D, C or Z, which takes a vector of type x as input and returns the index of the "largest" entry in absolute value, where in the complex case the size of z is $|\Re z| + |\Im z|$. The straightforward implementation of the reference BLAS [6] currently returns ISAMAX([0,NaN,2]) = 3 and ISAMAX([NaN,0,2]) = 1, which is clearly inconsistent, and does not always propagate a NaN as is our goal. Letting OV denote the overflow threshold, and letting $z_1 = .6 \cdot OV + i \cdot .6 \cdot OV$ and $z_2 = .7 \cdot OV + i \cdot .7 \cdot OV$, we get ICAMAX($[z_1, z_2]$) = 1 = ICAMAX($[z_2, z_1]$) because of overflow, even though both inputs are finite and differ in size.

For consistency, we propose requiring that IxAMAX always point to the first NaN if one exists, else to the first Inf, else to the first largest finite entry. Section 2.3.2 and Appendix A in [11] have correct implementations of ISAMAX and ICAMAX, resp. Appendix B gives test cases for ISAMAX and ICAMAX that are meant to stress test whether an implementation satisfies this specification. It also includes test results for a variety of existing BLAS implementations, both commercial and open-source, most of which exhibit failures for ISAMAX, and all of which do for ICAMAX (except our proposed new ones).

[1] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#Optimize-Options

Next consider GER, which computes a rank-1 update $A = A + \alpha \cdot x \cdot y^T$. The reference implementation checks for $y_i = 0$, and skips updating $A(:,i) = A(:,i) + \alpha \cdot x \cdot y_i$. In contrast, x is not checked for zero entries. Thus `Inf` and `NaN` in x and y are not propagated consistently.

The same issue arises in many other BLAS2 routines. In particular, consider TRSV, which solves $Tx = b$ for x where T is triangular. Suppose $T = U$ is upper triangular. Then the reference implementation will both check for trailing zeros in b, which lead to trailing zeros in x in the absence of exceptions, and for other zero entries of x that may occur because of cancellation. The code then skips multiplying column i of U by $x_i = 0$, so that `Inf`s and `NaN`s in column i of U are not propagated. In contrast, solving the same system by asking TRSV to solve $L^T x = b$ where $L = U^T$ does not do any checks for zeros.

Finally, we consider SGESV, 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 above 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 factor, 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 chooses 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`.

To deal with these inconsistencies, we propose *not* checking for zeros in situations like these. However, we do want to continue checking for zero scalars like α and β in operations like GEMM, $C = \alpha \cdot A \cdot B + \beta \cdot C$, and omitting either term if the scalar is zero. This is well-documented behaviour long expected by users, who, for example, want to compute $C = A \cdot B$ without needing to initialize C to make sure it doesn't contain `Inf`s or `NaN`s.

As an example where `NaN`s are anticipated and handled correctly, LAPACK routine SSTNEG counts the number of eigenvalues of a symmetric tridiagonal matrix that are less than a shift SIGMA, where the inner loop looks like:

```
DPLUS = D( J ) + T,
IF( DPLUS.LT.ZERO ) NEG1 = NEG1 + 1,
TMP = T / DPLUS,
T = TMP * LLD( J ) - SIGMA
```

A tiny DPLUS can cause T to overflow to `Inf`, which makes the next T = `Inf`/`Inf` = `NaN`, which propagates. Checking for this rare event in the inner loop would be expensive, so SLANEG only checks for T being a `NaN` every 128 iterations, recomputing the results more carefully, yielding significant speedups in the most common cases. The value 128 is a tuning parameter.

III. LAPACK - Proposed Exception Handling Interface

Our proposed fixes for the BLAS do not change the interface at all. In contrast, we do propose a new LAPACK interface for optionally reporting exceptions (as well as maintaining the existing "legacy" interface using a wrapper). Section III-A begins with a summary of all the user requests we received over the course of the design process. Section III-B summarizes our current design. Section III-C illustrates our design as applied to SGESV (a model implementation of the new SGESV, and all the routines in its call tree, appears in [11]).

A. User Requests

Our latest design (ninth in a sequence) tries to balance user requests ranging from not wanting to change any legacy code, to adding significant new exception handling capabilities in multithreaded environments, all with allowing the LAPACK developers to continue maintaining one core implementation. Specifically, we want to maintain just (1) one core that can be called from multiple languages, including C, C++, and Python, (2) one "wrapper" providing the legacy interface, and (3) the existing LAPACKE wrapper for C and C++ programmers.[2]

We start with a list of all the user requests we have received, and more thoughts about who may want control over exception handling and what they might want. These requests and related thoughts occurred at different points in our design process, which is why our design has gone through multiple versions:

(R1) I like my legacy code, calling LAPACK from Fortran/C/C++/NumPy, don't make me change it!

[2] https://netlib.org/lapack/lapacke.html

(R2) Ok if you want to help other folks with debugging, but don't slow down.

(R3) I'd like help debugging, when I find it necessary; I might need different kinds of information for this, depending on the situation, and just in the parts of the code where I suspect the problem to be. I'm willing to modify my code to do this, i.e. set the kind of exception handling I want, and get a report back, for each LAPACK call that I make.

(R4) I want to be able to set a flag at the beginning of execution that selects the kind of exception-handling I need for every LAPACK call and how to report them. I might want reporting done by returning information in a subroutine argument (analogous to LAPACK's INFO), or by collecting reports for multiple subroutine calls in some common data structure that I can inspect later. I don't want to modify my legacy code beyond setting this flag, and possibly inspecting the common data structure.

(R5) Same as (R4), except the flag should be settable (and changeable) at run-time.

(R6) Same as (R5), except I program in a multi-threaded/multi-task environment, so different threads/tasks may need to independently control how they handle and report exceptions, *i.e.*, depending on "context."

(R7) I want to write bullet proof code, i.e. that won't crash or give surprising wrong answers. I'm willing to slow down a little for this, but hopefully not much. All the approaches above from (R3) to (R6) are relevant.

(R8) I want to be able to turn off all error checking (e.g. $N < 0$) and exception handling, to run faster.

In addition to these requests, we considered which stakeholders might want to "control" the ways exceptions can be handled, including:

(C1) The user.

(C2) A library calling LAPACK internally.

(C3) Vendors of LAPACK equivalents.

(C4) Core LAPACK team, perhaps just prescribing what happens in "model" error handlers, with changes allowed in downstream customizations.

We want our interface design to be flexible enough to support all of these. Of course, if a user links versions of different routines that have been built with different assumptions about who is "in control," this could cause bugs which are beyond our control.

Regarding item C2 above, we considered the DOE xSDK (Extreme-scale Scientific Software Development Kit) project[3] which is a large LAPACK user, and maintains a set of "community policies" for library development.[4] Recommendation R3 and policies M11, M12, and M16 are particularly relevant. We briefly summarize these:

R3 Adopt and document a consistent system for propagating/returning error conditions/exceptions and provide an API for changing this behavior. (This is clearly consistent with our goals.)

M11 No hardwired print or I/O statements that cannot be turned off via an API. (This also impacts LAPACK's default use of XERBLA, which prints an error message and stops. Our design is independent of how XERBLA is implemented.)

M12 If a package imports software that is externally developed and maintained, then it must allow installing, building, and linking with an outside copy of that software. (This refers specifically to the BLAS and LAPACK.)

M16 Any xSDK-compatible package that compiles code should have a configuration option to build in Debug mode.

We believe that our design is consistent with this (long) list of requests and recommendations, and solicit comments.

As we considered the programming effort required to satisfy all these requests, we decided that we wanted there to be one core version of the LAPACK code to maintain that offers all these new features, and that can be called from all the different languages from which LAPACK is called, including C, C++, Python, and others. We also decided that the interface should be simple enough to allow significant code reuse across different LAPACK routines.

B. LAPACK Interface Proposal

Each LAPACK routine that already has an INFO argument will have a corresponding error-checking routine. The subroutine name will be changed to add _EC (for "error checking") to the end, allowing the original name to be retained providing the "legacy" interface and functionality.

In the new _EC version, following INFO (currently the last argument), 3 more arguments will be added:

1) FLAG_REPORT. For terseness and clarity, in the descriptions below we will use the abbreviations FLAG_REPORT(1) = WHAT (since it specifies what

[3] https://github.com/xsdk-project/

[4] https://github.com/xsdk-project/xsdk-community-policies/tree/master/package_policies

errors and exceptions to report), and FLAG_REPORT(2) = HOW (since it specifies how to report them).
2) INFO_ARRAY is an array used for more detailed reporting than possible using a single scalar INFO.
3) CONTEXT is an "opaque" argument that can be used to identify errors and exceptions associated with different threads or tasks.

(1) FLAG_REPORT: integer array, input only. We give a high level summary of the choices WHAT and HOW offer the user. The possible choices of WHAT errors or exceptions to report are as follows:

- WHAT ≤ -1: turn off all error checking.
- WHAT = 0: "legacy" error checking only.
- WHAT = 1: also check input and output arguments for `Infs` and `NaNs`.
- WHAT ≥ 2: also check input and output arguments throughout the call tree of the subroutine being called.

The user must independently choose HOW to report this information:

- HOW ≤ 0: only report using the scalar INFO.
- HOW = 1: also report more details using the array INFO_ARRAY.
- HOW = 2: also, if INFO $\neq 0$, call the routine REPORT_EXCEPTIONS, which can provide a customized way of reporting this information, e.g. a print statement, or recording this information in a data structure for later inspection. We will provide some simple model implementations of REPORT_EXCEPTIONS, but leave further customization to other software providers.
- HOW = 3: do all the above reporting throughout the call tree of the subroutine being called.
- HOW ≥ 4: call GET_FLAGS_TO_REPORT, to get values of WHAT and HOW, thus allowing users to choose WHAT and HOW with less modification of source code.

We note that the choice WHAT = HOW = 0 corresponds to the legacy LAPACK interface. We leave details of how other choices of WHAT and HOW interact with one another to [11].

(2) INFO_ARRAY: integer array, input/output. This is accessed only if WHAT ≥ 0 and HOW = 1, 2 or 3. The length of INFO_ARRAY is customized for each routinename, with detailed reporting as requested above, as follows:

- INFO_ARRAY(1) = legacy INFO.
- INFO_ARRAY(2) = value of FLAG_report(1) = WHAT that was used to determine the other entries of INFO_ARRAY.
- INFO_ARRAY(3) = value of FLAG_report(2) = HOW that was used to determine the other entries of INFO_ARRAY.
- INFO_ARRAY(4) = value of INFO depending on WHAT as described above.
- INFO_ARRAY(5) = number of routine arguments reported on.
- INFO_ARRAY(6) = number of internal LAPACK calls reported on.
- INFO_ARRAY(7:) contains a fixed number of entries, depending on the LAPACK routine, and on FLAG_report.

Here are more details on the values reported in INFO_ARRAY(7:).

Locations INFO_ARRAY(7:6+INFO_ARRAY(5)) contain one entry per floating point argument of the routine, with values:

- -1 if not checked (default).
- 0 if checked and ok (no `Inf` or `NaN` in input or output).
- 1 if it contains an input `Inf` or `NaN`, but not output.
- 2 if it contains an output `Inf` or `NaN`, but not input.
- 3 if it contains both an input and output `Inf` or `NaN`.

Input-only arguments are only checked on input, with possible return values in $\{0,1\}$, and output-only arguments are only checked on output, with possible return values in $\{0,2\}$. If an input argument has already been checked before calling the routine, this is indicated by setting INFO_ARRAY(*) = 0 on input (if checked and ok) or 1 (if it contains an `Inf` or `NaN`), otherwise INFO_ARRAY(*) should be set to -1 on input. For example, when calling SGESV_EC, the matrix A may have been checked by SGETRF_EC on output, so it does not need to be checked again by SGETRS_EC on input, saving work. Similarly, B may have been checked by SGESV_EC on input, so it does not need to be checked again by SGETRS_EC on input. Input values of INFO_ARRAY(*) less than -1 or greater than 1 will be treated the same as -1, i.e. not checked.

Locations INFO_ARRAY(7+INFO_ARRAY(5) : 6+INFO_ARRAY(5)+INFO_ARRAY(6)) contain one entry per LAPACK call (with an INFO parameter) appearing in the source code, with values:

- -1 if not checked (default).
- 0 if checked and ok.

- 1 if no input or output contains an `Inf` or `NaN`, but some LAPACK call deeper in the call chain signaled.
- 2 if an argument contains an input `Inf` or `NaN`, but not an output.
- 3 if an argument contains an output `Inf` or `NaN`, but not an input.
- 4 if an argument contains both an input and output `Inf` or `NaN`.

As before, we do not distinguish multiple calls to the same LAPACK routine at the same location (say inside a loop) in the source code, instead all their reports are combined into one, by taking the maximum of all the reporting values described above. Note that we do not attempt to report details about exceptions throughout the call chain of LAPACK routines. This is done by setting HOW = 3 so that the routine REPORT_EXCEPTIONS is called, as described below.

We note that INFO_ARRAY has a length customized for each routine. To make programming easier, we will document the maximum length of all these arrays, so that users can simply declare all INFO_ARRAY arrays to have this maximum length.

(3) CONTEXT: input only. This opaque argument may be accessed only if WHAT ≥ 0 and HOW = 2, 3 or 4.

When HOW = 2 or 3, it is used as an argument when LAPACK calls the routine REPORT_EXCEPTIONS (CONTEXT, SIZE_ROUTINENAME, ROUTINENAME, INFO_ARRAY) to report exceptional information in INFO_ARRAY immediately before returning from an LAPACK routine. REPORT_EXCEPTIONS will only be called if there is an exception to report, i.e. INFO is nonzero. CONTEXT is meant to accommodate application or architecture specific reporting methods, for example dealing with multithreaded programming environments, as in user request R6 (this argument can be ignored if it is not relevant). Here ROUTINENAME is a character array of length SIZE_ROUTINENAME, to indicate which routine is being reported on. All arguments are input only.

When HOW = 4, the LAPACK routine calls GET_FLAGS_TO_REPORT(CONTEXT, FLAG_REPORT) to get the (output-only) integer array FLAG_REPORT(1:2), which the user should have set by calling SET_FLAGS_TO_REPORT(CONTEXT, FLAG_REPORT) before calling the LAPACK routine with HOW = 4. This allows the user to report differently in different CONTEXTs. The default value of FLAG_REPORT should be [0, 0], i.e. legacy INFO reporting, in case the user has not called SET_FLAGS_TO_REPORT.

Since the semantics of CONTEXT are system dependent, we will only supply 2 placeholder versions of the 3 routines above, see [11] for details.

C. Example: SGESV_EC

We illustrate the proposed interface from the last section by summarizing how it applies to SGESV, yielding SGESV_EC. The calling sequence of SGESV_EC is as follows, where the 3 new arguments appear at the end:

SGESV_EC(N, NRHS, A, LDA, IPIV, B, LDB, INFO, FLAG_REPORT, INFO_ARRAY, CONTEXT)

First, we explain how to interpret INFO; the possible values are listed in decreasing priority order (only the first error found is reported):

1) "Legacy" values of INFO (WHAT ≥ 0):
   = 0 if successful execution, else
   = -1 if N < 0, else
   = -2 if NRHS < 0, else
   = -4 if LDA $<$ min(1,N), else
   = -7 if LDB $<$ min(1,N), else
   1 $\leq$ **INFO** $\leq$ **N** if U(INFO,INFO)=0, else ...
2) Possible values of INFO if checking input/output arguments for `Infs`/`NaNs` is requested (WHAT ≥ 1):
   = -3 if A contains an Inf/NaN on input, else
   = -6 if B contains an Inf/NaN on input, else
   = N+1 if A contains an Inf/NaN on output, else
   = N+2 if B contains an Inf/NaN on output, else ...
3) Possible values of INFO if checking internal subroutine calls for `Infs`/`NaNs` is requested (WHAT ≥ 2):
   = N+3 if SGETRF_EC had an Inf/NaN in an input/output, or a subroutine in its call tree did, else
   = N+4: ditto for SGETRS_EC.

Next, we explain how to interpret the entries of INFO_ARRAY, an array of length 10:

(1): INFO from "legacy" argument checking only.
(2): FLAG_REPORT(1) = WHAT to report.
(3): FLAG_REPORT(2) = HOW to report.
(4): INFO as determined by WHAT, as explained above
(5): ≤ 2; the number of arguments reported on (0 or 2, *i.e.*, A and B) .
(6): ≤ 2; the number of internal calls reported on (0 or 2, *i.e.*, SGETRF_EC and SGETRS_EC)
(7): Reports on `Infs`/`NaNs` in A (if WHAT ≥ 1):
   = -1 if not checked (default), else
   = 0 if checked and no `Infs`/`NaNs` on input/output, else

= **1** if checked and contains `Inf`/`NaN` on input but not output, else
= **2** if checked and contains `Inf`/`NaN` on output but not input, else
= **3** if checked and contains `Inf`/`NaN` on input and output.

(8): Ditto for B

(9): Reports on exceptions in call to SGETRF_EC (if WHAT $\geq$ 2):
= **-1** if not checked (default), else
= **0** if checked and no `Inf`s/`NaN`s, else
= **1** if checked and no input/output of SGETRF_EC contains an `Inf`/`NaN`, but some LAPACK call deeper in the call chain signalled an `Inf`/`NaN`, else
= **2** if checked and an input contains an `Inf`/`NaN`, but not an output, else
= **3** if checked and an output contains an `Inf`/`NaN`, but not an input, else
= **4** if checked and both an input and output contain an `Inf`/`NaN`.

(10): Ditto for call to SGETRS_EC.

IV. How to Test Consistency

We propose expanding our LAPACK test code to test both termination and error reporting. Recent work presented FPDiff [19], a differential testing framework for numerical functions which checks for correctness by discovering and comparing multiple implementations of the same function in different libraries. FPDiff discovered over 100 bugs in four numerical libraries that are attributable to a lack of unifying standards for exception handling. Similarly, approaches based on symbolic execution have been proposed to generate floating-point inputs that maximize error and trigger floating-point exceptions [20], [21], [22]. From these, [21] is the most relevant as it aims at triggering underflows and overflows, however their technique cannot be directly applied to Fortran code mainly due to a lack of support for symbolic execution of Fortran programs.

Random input generation has also been employed to generate floating-point inputs that mazimize error in the output of a program [23], [24]. In our case, we believe it is simplest to insert `Inf`s and `NaN`s 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 Section III. We propose to use *fuzzing* [25], which in our context means introducing an `Inf` or a `NaN` into one or more chosen variables during the course of an execution. We propose fuzzing because it is very difficult to devise an input without `Inf`s or `NaN`s 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 locations to introduce `Inf`s and `NaN`s, we can introduce them into subroutines called by the routine being tested to make sure that our reporting mechanisms work (or identify what needs to be fixed).

V. Conclusions

We have presented our proposed design to improve consistent exception handling in the BLAS and LAPACK, which could potentially also serve as a model for other numerical software. More details, including a draft implementation of SGESV_EC and all the routines in its call tree, and a list of the tasks (in priority order) needed to fully implement this proposal, are available in [11]. We invite user feedback.

Acknowledgements

This work was supported in part by the National Science Foundation under the project Basic ALgebra LIbraries for Sustainable Technology with Interdisciplinary Collaboration (BALLISTIC), Grant Nos. 2004763 (UC Berkeley), 2004541 (U. Tennessee) and 2004850 (U Colorado Denver), National Science Foundation Grant No. 1750983, and U.S. Department of Energy, Office of Science, Advanced Scientific Computer Research, under Grant No. DE-SC0020286. MathWorks also provided support.

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] T. Huckle and T. Neckel. *Bits and Bugs: A Scientific and Historical Review of Software Failures in Computational Science*. SIAM, 2019.

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

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

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

[9] 754-2019 IEEE Standard for Floating Point Arithmetic, 2019.
[10] 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.
[11] James Demmel, Jack J. Dongarra, Mark Gates, Greg Henry, Julien Langou, Xiaoye S. Li, Piotr Luszczek, Weslley Pereira, E. Jason Riedy, and Cindy Rubio-González. Proposed consistent exception handling for the BLAS and LAPACK. *CoRR*, abs/2207.09281, 2022.
[12] https://en.wikipedia.org/wiki/bfloat16_floating-point_format, 2017.
[13] grouper.ieee.org/groups/msc/ANSI_IEEE-Std-754-2019/background/minNum_maxNum_Removal_Demotion_v3.pdf, 2019.
[14] American National Standards Institute. *ANSI/ISO/IEC 9899-1999: Programming Languages — C*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1999.
[15] 2008 Fortran Standard, 2008. https://wg5-fortran.org/f2008.html.
[16] C11 standard, 2011. https://en.wikipedia.org/wiki/C11_(C_standard_revision).
[17] Programming languages – C, 2011. International Standard.
[18] Programming languages – C, ISO/IEC 9899:202x (E), February 5 2011. N2478 working draft.
[19] Jackson Vanover, Xuan Deng, and Cindy Rubio-González. Discovering discrepancies in numerical libraries. In *ISSTA*, pages 488–501. ACM, 2020.
[20] Hui Guo and Cindy Rubio-González. Efficient generation of error-inducing floating-point inputs via symbolic execution. In *ICSE*, pages 1261–1272. ACM, 2020.
[21] Earl T. Barr, Thanh Vo, Vu Le, and Zhendong Su. Automatic detection of floating-point exceptions. In *POPL*, pages 549–560. ACM, 2013.
[22] Roberto Bagnara, Matthieu Carlier, Roberta Gori, and Arnaud Gotlieb. Symbolic path-oriented test data generation for floating-point programs. In *ICST*, pages 1–10. IEEE Computer Society, 2013.
[23] Wei-Fan Chiang, Ganesh Gopalakrishnan, Zvonimir Rakamaric, and Alexey Solovyev. Efficient search for inputs causing high floating-point errors. In *PPoPP*, pages 43–52. ACM, 2014.
[24] Daming Zou, Ran Wang, Yingfei Xiong, Lu Zhang, Zhendong Su, and Hong Mei. A genetic algorithm for detecting significant floating-point inaccuracies. In *ICSE (1)*, pages 529–539. IEEE Computer Society, 2015.
[25] https://en.wikipedia.org/wiki/Fuzzing, July 2021.