Shterenlikht, A. (2019). On Quality of Implementation of Fortran 2008 Complex Intrinsic Functions on Branch Cuts. *ACM Transactions on Mathematical Software*, *45*(1), [11]. https://doi.org/10.1145/3301318

On quality of implementation of Fortran 2008 complex intrinsic functions on branch cuts

A. SHTERENLIKHT, The University of Bristol, UK

Branch cuts in complex functions have important uses in fracture mechanics, jet flow and aerofoil analysis. This paper introduces tests for validating Fortran 2008 complex functions - LOG, SQRT, ASIN, ACOS, ATAN, ASINH, ACOSH and ATANH - on branch cuts with arguments of all 3 IEEE floating point binary formats: binary32, binary64 and binary128, including signed zero and signed infinity. Multiple test failures were revealed, e.g. wrong signs of results or unexpected overflow, underflow, or NaN. We conclude that the quality of implementation of these Fortran 2008 intrinsics in many compilers is not yet sufficient to remove the need for special code for branch cuts. The electronic appendix contains the full test results with 8 Fortran 2008 compilers: GCC, Flang, Cray, Oracle, PGI, Intel, NAG and IBM, detailed derivations of the values of these functions on branch cuts and conformal maps of the branch cuts, to be used as a reference. The tests and the results are freely available from https://cmplx.sourceforge.io. This work will be of interest to engineers who use complex functions, as well as to compiler and math library developers.

Additional Key Words and Phrases: Fortran, LOG, SQRT, ASIN, ACOS, ATAN, ASINH, ACOSH, ATANH, branch cuts, signed zero, signed infinity

ACM Reference Format:
A. Shterenlikht. 2018. On quality of implementation of Fortran 2008 complex intrinsic functions on branch cuts. 1, 1 (November 2018), 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In the following $w = u + iv$ and $z = x + iy$ are complex variables, $w = f(z)$ is a conformal mapping function from z to w and $z = f^{-1}(w)$ is a conformal mapping function from w to z. $\Re z$ and $\Im z$ are the real and the imaginary parts of z.

Complex functions with branch cuts have useful applications e.g. in fracture mechanics, because a branch cut can represent a mathematical crack. Perhaps the oldest and simplest example is function

$$z = w + 1/w \tag{1}$$

which maps a complex plane with a cut unit circle onto a complex plane with a cut along x at $-2 \le x \le 2$. This function has been in use probably since early 20th century, see e.g. [17, 21]. It is still widely used in fracture mechanics today [18]. In practice the inverse of Eqn. (1) is more useful:

$$w = \frac{1}{2}\left(z + \text{copySign}(1, \Re z)\sqrt{z^2 - 4}\right) \tag{2}$$

where *copySign* is the IEEE function which returns a value with the magnitude of the first argument and the sign of the second argument [13]. The map of Eqn. (2) is shown in Fig. 1.

Author's address: A. Shterenlikht, The University of Bristol, Department of Mechanical Engineering, Queen's Building, University Walk, Bristol, BS8 1TR, UK, mexas@bristol.ac.uk.

XXXX-XXXX/2018/11-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

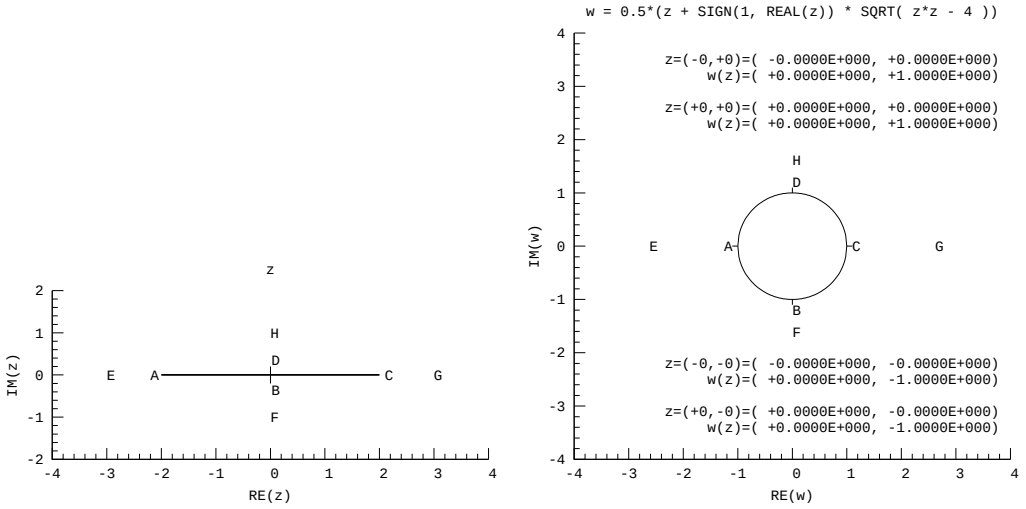

Fig. 1. Map of $w = \frac{1}{2}(z + \text{copySign}(1, \Re z)\sqrt{z^2 - 4})$. The branch cut ABCD in z is mapped onto a unit circle ABCD in w.

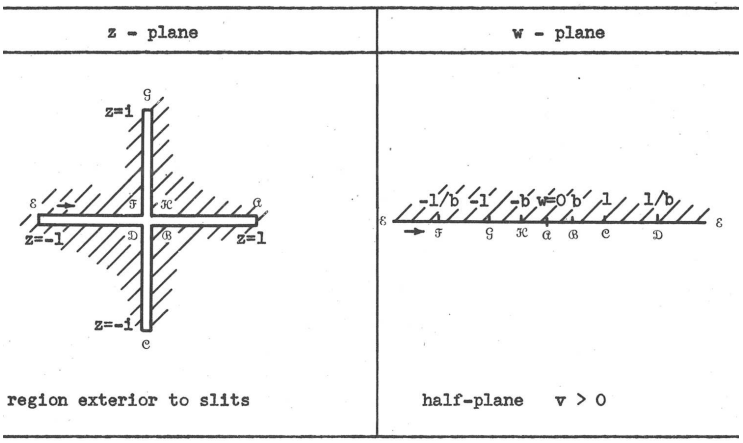

Fig. 2. Map of function $w = \tan(\arccos z^2/4)$, reproduced from [17].

Note that Eqn. (2) produces the desired mapping only if $+0$ and -0 can be distinguished, so that points in z on the top and the bottom boundary of the cut, i.e. with $y = +0$ and $y = -0$ are mapped respectively onto the top and the bottom boundary of the unit circle in w. For example, point $z = +0 - \mathrm{i}0$ is mapped to point $w = +0 - \mathrm{i}1$, point B in Fig. 1, and point $z = +0 + \mathrm{i}0$ is mapped to point $w = +0 + \mathrm{i}1$, point D in Fig. 1.

Another function, useful for the study of intersecting cracks, is $w = \tan(\arccos z^2/4)$ [17, p. 79], which maps a plane with 2 intersecting cuts onto an upper half plane, $v \geq 0$. The two cuts form a cross centered at the origin, see Fig. 2. Two branch cuts in arccos along the real axis together with the ability to distinguish $+0$ and -0, mean that points B, D, F and H, located at the origin in z are mapped onto 4 distinct points in w in Fig. 2.

In fact, there are 8 elementary complex functions with branch cuts – log, $\sqrt{}$, three inverse trigonometrics (arcsin , arccos, arctan) and three inverse hyperbolic functions (arcsinh , arccosh, arctanh) – all of which have useful applications in fracture and aerodynamics [15, 16]. For example, log has a single branch cut along the negative real axis. Therefore it can be used for analysis of an edge crack in an infinite plate. arcsin , arccos, arctan, arcsinh and arctanh have 2 cuts on either the real or the imaginary axis, and can therefore be used for the analysis of bodies with 2 cracks along the same line, e.g. an infinite or a finite width plate with 2 opposing cracks with a finite ligament length in between. This case is of significant practical importance in fracture mechanics, see e.g. [23, Sec. 4, 'Parallel Cracks']. arccosh has a single branch cut and can be used for an edge crack geometry.

In all these 8 functions the cuts lie either along the real axis, $x = 0$, or along the imaginary axis, $y = 0$. Hence, the ability to distinguish $+0$ and -0 is required in applications of these elementary functions in science and engineering, so that the sides of each cut can be mapped independently. Jet flows and aerofoils are among other popular practical examples where signed zero is required to obtain correct conformal maps of multivalued complex functions on branch cuts [15, 16]. The usage of -0 was further popularised, although with no new examples, in [4, 22]. It is important to note that signed zero, ± 0, is linked to signed infinities, $\pm\infty$, e.g. $\frac{1}{+0} = +\infty$ but $\frac{1}{-0} = -\infty$. Hence the use of complex intrinsics with branch cuts for science and engineering applications needs support for signed infinity too.

The IEEE floating point standard [7] defined signed zero and signed infinity: $+0, -0, +\infty, -\infty$, as early as 1985. Expressions for these 8 complex intrinsics, which deal correctly with ± 0, $\pm\infty$ and NaN, and avoid cancellation, were given by W. Kahan in 1987 [15]. A recent study concludes that no better expressions have been proposed since then [20]. However, to date support for ± 0 and $\pm\infty$ in math libraries is varied. If signed zero or signed infinity are not available, algorithms can be, and have been, developed which use data a short distance away from the cuts. However, this is not very satisfactory, as it is not obvious what this small distance should be. In addition, branch cuts often contain the most important data, e.g. the extremum values of crack tip displacement fields are found on crack flanks, which is useful in experimental fracture mechanics analysis [19]. It would help algorithm developers and programmers significantly if they had full confidence that complex functions behave correctly on branch cuts, and no special cases need to be considered and coded for.

Given that Fortran is still the most widely used language in science and engineering, particularly in high performance computing, where Fortran codes use 60-70% of machine cycles [24], we focus on implementation of the above 8 complex functions in Fortran. For C programmers we note that specifications for complex math functions for ± 0, $\pm\infty$ and NaN were added in C99 [10].

In the following, Fortran functions and written in `MONOSPACE UPPERCASE` and all other Fortran names are written in `monospace lowercase` font.

The Fortran intrinsic functions `SQRT` and `LOG` have accepted complex arguments at least since the FORTRAN66 standard [1]. The Fortran 2003 standard [8] added support for the IEEE floating point arithmetic. Fortran 2008 standard [9] added support for complex arguments to intrinsic functions `ACOS`, `ASIN`, `ATAN` and 3 new inverse hyperbolic intrinsics: `ACOSH`, `ASINH`, `ATANH`, all of which also accept complex arguments. With Fortran 2008, programmers finally have access to intrinsics implementing the above 8 elementary complex functions, including on the branch cuts. However, the question of how well the above 8 complex functions are implemented in modern Fortran still deserves attention. This question is addressed in this work with the introduction of a set of 96 tests, which check correctness of the 8 Fortran 2008 complex intrinsics on branch cuts. The code used in this work is freely available from https://cmplx.sourceforge.io.

2 TESTS

The tests are designed to verify the behaviour of the 8 intrinsic Fortran functions at special points on branch cuts. All three IEEE basic binary formats are verified: binary32, binary64 and binary128 [13]. To aid portability, the Fortran 2008 intrinsic module `iso_fortran_env` includes named constants for these IEEE data formats: `REAL32`, `REAL64` and `REAL128`, which are used to define the kinds of real and complex variables and constants in the tests as e.g:

```
use, intrinsic :: iso_fortran_env
integer, parameter :: fk=real64
real(kind=fk), parameter :: one=1.0_fk
```

The tests check that the signs of the real and the imaginary parts are correct, and that no undue overflow, underflow or NaN results are produced. The Fortran IEEE intrinsics `IEEE_CLASS`, `IEEE_COPY_SIGN`, `IEEE_IS_FINITE`, `IEEE_IS_NAN`, `IEEE_SUPPORT_SUBNORMAL`, `IEEE_SUPPORT_INF`, `IEEE_SUPPORT_NAN`, `IEEE_VALUE` are used, as well as the named constants `ieee_negative_inf`, `ieee_positive_inf`, `ieee_negative_zero`, `ieee_positive_zero`, `ieee_positive_denormal` and `ieee_negative_denormal`. For example, the values of ± 0 and $\pm\infty$ are defined in the tests as:

```
real(kind=fk) :: infp, infm, zerop, zerom
 infp=IEEE_VALUE( one, ieee_positive_inf )
 infm=IEEE_VALUE( one, ieee_negative_inf )
zerop=IEEE_VALUE( one, ieee_positive_zero )
zerom=IEEE_VALUE( one, ieee_negative_zero )
```

In addition, the Fortran intrinsics `HUGE`, `TINY` and `EPSILON` are used, which return the largest and the smallest positive model (normalised) numbers respectively, here denoted h and t, and machine epsilon, ϵ. Note that the Fortran definition of ϵ is $\epsilon = r^{1-p}$, where r is the radix, $r = 2$ on binary computers, and p is the precision. This definition follows the IEEE standard [13].

The accuracy of complex floating point calculations has been analysed in a number of works. Expressions for the relative errors of complex $\sqrt{}$ and log (as well as exp, sin, cos) are given in [5], although the authors did not distinguish $+0$ and -0. The expressions are given in terms of the relative errors of the real counterparts of these intrinsics, e.g. their bound for the relative error in complex $\sqrt{}$ is $2\epsilon + 1.5E_{\text{sqrt}}$, where E_{sqrt} is the relative error bound for real $\sqrt{}$. [3] proposed a high speed implementation of complex $\sqrt{}$ which preserved the accuracy of [5]. For complex log [5] gives the relative error bound of $3.886\epsilon + E_{\log}$, where $E_{\log}$ is the relative error bound for real $\log x$, $x \gg 1$. For arcsin and arccos [6] give the relative error bound of 9.5ϵ. The relative error bound of a fused multiply-add (FMA) for complex multiplication was recently estimated as low as ϵ [14].

Based on these accuracy estimates, in this work a conservative relative error bound of $10^2\epsilon$ was considered acceptable for π, $\pi/2$ and 1, the magnitudes of the real or the imaginary parts of the result values on the branch cuts. Another reason for choosing a high error bound is that the Fortran 2008 standard is deliberately vague about the accuracy of floating point intrinsics, e.g. for log on the branch cut it just says that the imaginary part of the result is 'approximately π' or 'approximately $-\pi$', depending on the cut side [9].

Where the real or the imaginary part of the result is predicted analytically to be ± 0, it was validated against `zerop` or `zerom` respectively, i.e. exact result values were expected for ± 0.

Although C11 [11, App. G.6] specifies the return values of these 8 complex intrinsics on the branch cuts, including points at infinity, and details which exceptions should be raised, Fortran 2008 has no such constraints [9, Clause 13.7]. Therefore, the electronic appendix contains concise but full derivations of analytic expressions for the 8 intrinsics on the branch cuts, including points

at infinity. These expressions are used as a reference to validate the values returned by the Fortran 2008 intrinsics.

A summary of the test results is given in Sec. 3. The detailed results can be found in the electronic appendix, together with the reference conformal maps of the branch cuts for the 8 intrinsics. The reader can use the maps, which are similar to Fig. 1, as a graphical aid in visualising the locations of the test points.

In the following, where ± occurs in both the argument and the result, the result has the same sign as the argument.

2.1 LOG

The behaviour of `LOG` was checked on the branch cut at 8 points: $z = -\infty \pm \mathrm{i}0$, $z = -h \pm \mathrm{i}0$, $z = -1 \pm \mathrm{i}0$ and $z = -t \pm \mathrm{i}0$. The top and the bottom boundaries of the cut are mapped to $w = u + \mathrm{i}\pi$ and $w = u - \mathrm{i}\pi$ respectively.

2.2 SQRT

The behaviour of `SQRT` was checked on the branch cut at 10 points: $z = -\infty \pm \mathrm{i}0$, $z = -h \pm \mathrm{i}0$, $z = -1 \pm \mathrm{i}0$, $z = -t \pm \mathrm{i}0$ and $z = -0 \pm \mathrm{i}0$. The top boundary of the cut is mapped onto the positive imaginary axis, and the bottom boundary of the cut is mapped onto the negative imaginary axis.

2.3 ASIN

The behaviour of `ASIN` was checked on 12 points: $z = \pm\infty \pm \mathrm{i}0$, $z = \pm h \pm \mathrm{i}0$ and $z = \pm 1 \pm \mathrm{i}0$. $w = \arcsin z$ maps a plane with 2 cuts along the real axis, $x \le -1$ and $x \ge 1$ to an infinite strip of width π along the imaginary axis, $-\pi/2 \le u \le \pi/2$. The left cut, $x \le -1$ is mapped onto the left boundary of the strip, $u = -\pi/2$. The right cut, $x \ge 1$ is mapped onto the right boundary of the strip, $u = \pi/2$.

2.4 ACOS

The behaviour of `ACOS` was checked on the same 12 points as of `ASIN`. $w = \arccos z$ has 2 branch cuts, both on the real axis, at $x \le -1$ and $x \ge 1$. For $x \le -1$, the top boundary of the cut, $y = +0$, is mapped to $w = \pi - \mathrm{i}b$ and the bottom boundary of the cut, $y = -0$, is mapped to $w = \pi + \mathrm{i}b$. For $x \ge 1$, the top boundary of the cut, $y = +0$, is mapped to $w = +0 - \mathrm{i}b$, and the bottom boundary of the cut, $y = -0$, is mapped to $w = +0 + \mathrm{i}b$. In all cases $b \ge 0$.

2.5 ATAN

The behaviour of `ATAN` was checked on 16 points: $z = \pm 0 \pm \mathrm{i}\infty$, $z = \pm 0 \pm \mathrm{i}h$, $z = \pm 0 \pm \mathrm{i}1$ and $z = \pm 0 \pm \mathrm{i}(1 + \epsilon)$. The last 4 values are interesting because they are likely to be used as the best substitute for $\pm 0 \pm \mathrm{i}1$ on systems which do not support $\pm\infty$. $w = \arctan z$ maps a plane with 2 cuts along the imaginary axis, $y \le -1$ and $y \ge 1$ to an infinite strip along the imaginary axis of width π and centred on zero.

Note that $\Im \arctan(\pm 0 \pm \mathrm{i}h)$ is subnormal (C11 uses the term *subnormal* instead of the earlier *denormal*), e.g. for `REAL64` the smallest normal number is $\approx 2.2 \times 10^{-308}$ while $|\Im \arctan(\pm 0 \pm \mathrm{i}h)| \approx 5.6 \times 10^{-309}$ (see the electronic appendix for full details). On systems with no support for subnormals the correct result is $\Im \arctan(\pm 0 \pm \mathrm{i}h) = \pm 0$, with the correct sign. On the other hand, on systems with no support for subnormals, a subnormal return value is not acceptable, because such value, k, would violate the expected inequalities $|k| > 0$ and $|k| < t$ [9].

C11 defines $\arctan(\pm 0 \pm \mathrm{i}1) = \pm 0 \pm \mathrm{i}\infty$ [11, Annex G.6]. The expressions given in the electronic appendix are different: $\arctan(\pm 0 \pm \mathrm{i}1) = \pm\pi/2 \pm \mathrm{i}\infty$. However, it is easy to show [2, Eqn. 4.21.39] that a more relaxed expression: $\arctan(\pm 0 \pm \mathrm{i}1) = \pm q \pm \mathrm{i}\infty$, where $q = +0$ or $0 < q \le \pi/2$,

is sufficient to satisfy the identity $\tan(\arctan z) = z$. Hence the tests use the relaxed expression above to validate $\Re \arctan(\pm 0 \pm \mathrm{i}1)$.

2.6 ASINH

The behaviour of ASINH was checked on 12 points: $z = \pm 0 \pm \mathrm{i}\infty$, $z = \pm 0 \pm \mathrm{i}h$ and $z = \pm 0 \pm \mathrm{i}1$. $w = \operatorname{arcsinh} z$ maps a plane with 2 cuts along the imaginary axis, $y \le -1$ and $y \ge 1$ to an infinite strip of width π along the real axis, $-\pi/2 \le v \le \pi/2$. The bottom cut, $y \le -1$ is mapped onto the bottom boundary of the strip, $v = -\pi/2$. The top cut, $y \ge 1$ is mapped onto the top boundary of the strip, $v = \pi/2$.

2.7 ACOSH

The behaviour of ACOSH was checked on 10 points: $z = -\infty \pm \mathrm{i}0$, $z = -h \pm \mathrm{i}0$, $z = -1 \pm \mathrm{i}0$, $z = +0 \pm \mathrm{i}0$ and $z = 1 \pm \mathrm{i}0$. $w = \operatorname{arccosh} z$ maps a plane with a single cut along the real axis at $x \le 1$ onto a semi-infinite strip of width 2π, running along the real axis, $u \ge 0$. The tests check that (1) the top side of the cut at $x \le -1$ is mapped onto the top boundary of the strip, $u = +0$ and $u > 0, v = \pi$; (2) the top side of the cut at $-1 \le x \le 1$ is mapped onto the end of the strip at $u = +0, v = +0$ and $0 < v \le \pi$; (3) the bottom side of the cut at $-1 \le x \le 1$ is mapped onto the end of the strip at $u = +0, v = -0$ and $-\pi \le v < 0$, and (4) the bottom side of the cut at $x \le -1$ is mapped onto the bottom boundary of the strip, $u = +0$ and $u > 0, v = -\pi$.

2.8 ATANH

ATANH was verified on 16 points: $z = \pm\infty \pm \mathrm{i}0$, $z = \pm h \pm \mathrm{i}0$, $z = \pm 1 \pm \mathrm{i}0$ and $z = \pm(1 + \epsilon) \pm \mathrm{i}0$. $w = \operatorname{arctanh} z$ maps a plane with 2 cuts along the real axis, $x \le -1$ and $x \ge 1$ onto a infinite strip of width π centered on 0 and running along the real axis.

The behaviour of ATANH on the branch cuts mirrors many features of that of ATAN, since $\arctan z = -\mathrm{i}\operatorname{arctanh}(\mathrm{i}z)$. C11 defines $\operatorname{arctanh}(\pm 1 \pm \mathrm{i}0) = \pm\infty \pm \mathrm{i}0$ [11, Annex G.6]. The expressions given in the electronic appendix are different: $\operatorname{arctanh}(\pm 1 \pm \mathrm{i}0) = \pm\infty \pm \mathrm{i}\pi/2$. However, it is easy to show, using [2, Eqn. 4.35.36], that a more relaxed expression: $\operatorname{arctanh}(\pm 1 \pm \mathrm{i}0) = \pm\infty \pm \mathrm{i}q$, where $q = +0$ or $0 < q \le \pi/2$, is sufficient to satisfy the identity $\tanh(\operatorname{arctanh} z) = z$. Hence the tests use the relaxed expression above to validate $\Im \operatorname{arctanh}(\pm 1 \pm \mathrm{i}0)$.

Clearly the same q must be taken for $\arctan(\pm 0 \pm \mathrm{i}1) = \pm q \pm \mathrm{i}\infty$ and for $\operatorname{arctanh}(\pm 1 \pm \mathrm{i}0) = \pm\infty \pm \mathrm{i}q$, for the identity $\arctan z = -\mathrm{i}\operatorname{arctanh}(\mathrm{i}z)$ to hold.

3 SUMMARY OF THE RESULTS AND DISCUSSION

The detailed test results are given in the electronic appendix and at https://cmplx.sourceforge.io. The main conclusion is that the quality of implementation varies significantly between the 8 compilers tested.

Most compiler documentation referred to during this work indicates that evaluation of the 8 complex intrinsics is done via external calls, typically to `libm`. Therefore, the diversity of results between compilers is surprising. Although in some cases identical failures are seen, e.g. with Cray and Oracle for arcsinh for REAL32 and REAL64 kinds, or with Cray and GCC for REAL128 kind for all intrinsics, in general different failure patterns are seen in each compiler. This indicates that not all vendors use the same algorithms and/or math libraries.

Only a single compiler has passed all 96 tests for all 3 IEEE floating point types. Another compiler has passed all 96 tests for REAL32 and REAL64 kinds.

As mentioned in the introduction, both LOG and SQRT Fortran intrinsics accepted complex arguments at least as far back as FORTRAN66, and perhaps even earlier. Therefore it was surprising to find that one compiler failed several log tests, and 4 out of 8 compilers showed multiple failures

in $\sqrt{}$ tests with REAL32 and REAL64 kinds, including overflow, wrong sign and NaN. Given that all CPUs used in this work are meant to fully support IEEE arithmetic with REAL32 and REAL64 kinds (except possibly support for subnormals, which might be implemented in software) and had hardware instructions for single and double precision $\sqrt{}$, we speculate that the problems are likely in compiler implementations of complex $\sqrt{}$.

Many failures of type "*n*", were obtained. These are failures where NaN values were produced. None of these 8 intrinsics should produce NaN results on branch cuts, including points at infinity. Hence, such failures are obviously completely unacceptable. This is the most obvious failure type, both to the programmer and to the compiler or library developers. The vendors should be able to find and fix all such failures easily.

Another frequently observed failure type was "*o*", overflow, i.e. when $\pm\infty$ results were produced instead of the correct finite values. These are most likely caused by overflow in the intermediate computations in the math library. These failures are more dangerous to the programmer, because they can be hidden by consecutive calculations.

In our opinion the most dangerous type of failure to the programmer is type "*s*", where the sign of the real or the imaginary part of the result, or both, is wrong. Such failures will likely cause unexpected results further down in the calculations, which will be hard to debug. Expressions carefully derived in the electronic appendix are intended as a reference and a debugging aid.

Other failure types were seen less often. Failure of type "*z*", where a zero result was obtained instead of the correct non-zero normal value was seen only together with other failure types, overflow and NaN. We therefore recommend the vendors to focus on resolving failure types "*n*" and "*o*" first. Failures of types "*d*", where a subnormal result was obtained while the processor did not support subnormals, were seen only in a single compiler. Likewise, failures of type "*m*", where the magnitude of the real or the imaginary part was clearly wrong, were peculiar to a single vendor.

Finally, a single vendor erroneously *printed* +0 in formatted output for −0 internal representation. Since the tests are currently done using only the internal representations of the result values, and not the printed values, such compiler behaviour did not result in test failure. However, the users reading the wrongly signed zero values in print can be misled. Hence, we flag such tests as "*g*", to alert the user.

It is important to emphasise that only failures of type "*n*", where NaN results were produced, can be interpreted as compiler non-conformance with the standard. This is because Fortran 2008, or any previous Fortran standard, requires very little in terms of accuracy of floating point calculations. Descriptions of many intrinsics have only the phrase 'processor-dependent approximation', e.g. the result of arccosh(X) is defined as 'a value equal to a processor-dependent approximation to the inverse hyperbolic cosine function of X', where 'processor' is defined as a 'combination of a computing system and mechanism by which programs are transformed for use on that computing system' [9], i.e. it includes the compiler, the libraries, but also the runtime environment and the hardware. Therefore, we interpret the test results only as 'quality of implementation'.

4 RECOMMENDATIONS FOR A FUTURE FORTRAN STANDARD

Fortran 2008 and the draft 2018 standards [9, 12] prohibit LOG from accepting a zero argument, likely because the imaginary part of $\log(\pm 0 \pm \mathrm{i}0)$ is mathematically undefined. It is proposed that future Fortran standards allow $\log(\pm 0 \pm \mathrm{i}0)$ with the return values used by C11 [11, Annex G.6]:

$$\log(-0 + \mathrm{i}0) = -\infty + \mathrm{i}\pi; \qquad \log(+0 + \mathrm{i}0) = -\infty + \mathrm{i}0; \qquad \log(\mathrm{conj}(z)) = \mathrm{conj}(\log(z)) \tag{3}$$

Allowing $\log(\pm 0 \pm \mathrm{i}0)$ would be useful to the programmer, because it will make the fundamental identity $z^a = \exp(a \log z)$ valid for all z. An immediately useful example is $\sqrt{-0 \pm \mathrm{i}0}$:

$$\sqrt{-0 \pm \mathrm{i}0} = \exp(\frac{1}{2}\log(-0 \pm \mathrm{i}0)) = \exp(\frac{1}{2}(-\infty \pm \mathrm{i}\pi)) = \exp(-\infty)(\cos\frac{\pi}{2} \pm \mathrm{i}\sin\frac{\pi}{2}) = +0 \pm \mathrm{i}0 \quad (4)$$

5 CONCLUSIONS

96 tests for complex Fortran 2008 intrinsics `LOG`, `SQRT`, `ACOS`, `ASIN`, `ATAN`, `ACOSH`, `ASINH` and `ATANH` on branch cuts were designed for this work. Only 2 compilers passed all tests with IEEE binary32 and binary64 types and only a single compiler passed all tests with all 3 IEEE floating point types. Based on this limited testing, the user is advised to deploy inverse trigonometric and hyperbolic intrinsics, $\sqrt{}$ and log on branch cuts with caution, using extensive testing of the algorithms on known cases. Unfortunately the need to use special code for calculations on branch cuts has not yet disappeared completely. We expect the quality of implementation in all compilers to improve in line with customer demands. The immediate future work will include checks for exceptions, and also for additional IEEE capabilities added in the Fortran 2018 standard. Finally, we welcome any feedback on our tests, such as bug reports or results from other compilers or compiler versions. These can be submitted via https://cmplx.sourceforge.io.

6 ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

We acknowledge the use of several computational facilities for this work: The ARCHER UK National Supercomputing Service, project eCSE05-05; Advanced Computing Research Centre of The University of Bristol and The STFC Hartree Centre. The STFC Hartree Centre is a research collaboration in association with IBM providing High Performance Computing platforms funded by the UK's investment in e-Infrastructure.

REFERENCES

[1] ANSI X3.9-1966. 1966. *FORTRAN Standard.*
[2] DLMF 2018. NIST Digital Library of Mathematical Functions. Release 1.0.19 of 2018-06-22. http://dlmf.nist.gov/ F. W. J. Olver, A. B. Olde Daalhuis, D. W. Lozier, B. I. Schneider, R. F. Boisvert, C. W. Clark, B. R. Miller and B. V. Saunders, eds.
[3] M. D. Ercegovac and J.-M. Muller. 2007. Complex square root with operand prescaling. *Journal of VLSI signal processing* 49, 1 (Oct. 2007), 19–30. https://doi.org/10.1007/s11265-006-0029-2
[4] D. Goldberg. 1991. What every computer scientist should know about floating-point arithmetic. *Comput. Surveys* 23, 1 (March 1991), 5–48. https://doi.org/10.1145/103162.103163
[5] T. E. Hull, T. F. Fairgrieve, and P. T. P. Tang. 1994. Implementing Complex Elementary Functions Using Exception Handling. *ACM Transactions of Mathematical Software* 20, 2 (Dec. 1994), 215–244. https://doi.org/10.1145/178365.178404
[6] T. E. Hull, T. F. Fairgrieve, and P. T. P. Tang. 1997. Implementing the complex arcsine and arccosine functions using exception handling. *ACM Transactions of Mathematical Software* 23, 3 (Sept. 1997), 299–335. https://doi.org/10.1145/275323.275324
[7] IEEE Std 754-1985. 1985. *IEEE Standard for Floating-Point Arithmetic.*
[8] ISO/IEC 1539-1:2004. 2004. *Information technology – Programming languages – Fortran – Part 1: Base language.*
[9] ISO/IEC 1539-1:2010. 2010. *Information technology – Programming languages – Fortran – Part 1: Base language.*
[10] ISO/IEC 9899:1999. 1999. *Programming languages – C.*
[11] ISO/IEC 9899:2011. 2011. *Information technology – Programming languages – C.*
[12] ISO/IEC DIS 1539-1. 2018. *Information technology – Programming languages – Fortran – Part 1: Base language.*
[13] ISO/IEC/IEEE 60559:2011. 2011. *Information technology – Microprocessor Systems – Floating-Point arithmetic.*
[14] C.-P. Jeannerod, P. Kornerup, N. Louvet, and J.-M. Muller. 2017. Error bounds on complex floating-point multiplication with an FMA. *Math. Comp.* 86, 304 (March 2017), 881–898. https://doi.org/10.1090/mcom/3123

[15] W. Kahan. 1987. Branch Cuts for Complex Elementary Functions, or Much Ado About Nothing's Sign Bit. In *The State of the Art in Numerical Analysis*, A. Iserles and M. J. D. Powell (Eds.). Clarendon Press, Oxford.
[16] W. Kahan. 1997. The John von Neumann lecture. In *45th SIAM annual meeting*. https://people.eecs.berkeley.edu/~wkahan/SIAMjvnl.pdf
[17] H. Kober. 1952. *Dictionary of Conformal Representations*. Dover.
[18] P. Lopez-Crespo, R. L. Burguete, E. A. Patterson, A. Shterenlikht, P. J. Withers, and J. R. Yates. 2009. Study of a Crack at a Fastener Hole by Digital Image Correlation. *Experimental Mechanics* 49, 4 (Aug. 2009), 551–559. https://doi.org/10.1007/s11340-008-9161-1
[19] P. Lopez-Crespo, A. Shterenlikht, E. A. Patterson, J. R. Yates, and P. J. Withers. 2008. The stress intensity of mixed mode cracks determined by digital image correlation. *Journal of Strain Analysis for Engineering Design* 43, 8 (2008), 769–780. https://doi.org/10.1243/03093247JSA419
[20] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. 2010. *Handbook of Floating-Point Arithmetic*. Birkhäuser, Boston.
[21] N. I. Muskhelishvili. 1953. *Some Basic Problems of the Mathematical Theory of Elasticity; translated from the Russian by J.R.M. Radok* (3 ed.). Groningen, Noordhoff.
[22] M. L. Overton. 2001. *Numerical Computing with IEEE Floating Point Arithmetic*. SIAM.
[23] H. Tada, P. C. Paris, and G. R. Irwin. 2000. *The Stress Analysis of Cracks Handbook* (3 ed.). ASME.
[24] A. Turner. 2015. Parallel software usage on UK national HPC facilities 2009-2015. ARCHER white papers, EPCC, Edinburgh, UK. http://archer.ac.uk/documentation/white-papers/app-usage/UKParallelApplications.pdf