Handling IEEE 754 Invalid Operation Exceptions in Real Interval Arithmetic

Douglas M. Priest

Draft revised May 12, 1997

Abstract

Interval arithmetic operations implemented using IEEE floating point can deliver incorrect results when certain exceptions are mishandled. This note proposes a way to handle those exceptions correctly. We first define a set of representable real intervals based on the IEEE floating point number system. We then show that we can implement consistent arithmetic on these intervals with efficient algorithms that deliver correct results even when exceptions occur. Finally, we suggest ways to incorporate these ideas into improved hardware support for interval arithmetic.

1 Introduction

How reliable are the results of interval computations? The expression $ab[(1/a)^2 - (1/b)^2]$ simplifies to $3/2$ when $b = 2a$, but a simple program using INTLIB[7] to compute an interval bound on this expression for $a = 10^{-175}$ and $b = 2 \cdot 10^{-175}$ obtains the interval $[-2.23 \cdot 10^{-308}, 2.23 \cdot 10^{-308}]$ on many systems. The quantity

$$y\left(\frac{1}{-(1/x-1)^2}+1\right)$$

is unbounded when x ranges over the interval $[10^{-310}, 1]$ and y ranges over the interval $[-1, 1]$, but a straightforward program using Knüppel's BIAS subroutine library[9] to compute a bound on this quantity for x and y in those intervals produces the result $[-1, 1]$ on at least three different machines. The expression $1/(s^2t^2+1)$ can take values extremely close to one when s ranges over the interval $[10^{-200}, 1]$ and t ranges over the interval $[1, 10^{200}]$, but a short program to compute a bound on this expression using

the interval arithmetic incorporated in Van Iwaarden's VerGO software[12] delivers the interval $[0, 1/2]$ on many computers.

The preceding results were all obtained on systems conforming to the IEEE 754 standard[2]. In each case, the computation produced an incorrect result because the software implementing the basic interval arithmetic operations did not account for a situation defined by the standard as an *invalid operation exception*. In general, an exception occurs whenever a floating point operation has no single universally acceptable result, so for each exception, the standard prescribes a default result that may depend on one or more modes established by the program. Two of these modes, the round-to-positive-infinity mode and the round-to-negative-infinity mode, are intended to support interval arithmetic. In these modes, the default results for exceptional operations are defined to give either the best representable upper or lower bound on the true result or else an indication that no such bound can be delivered.

Unfortunately, the default result specified by the IEEE standard for an invalid operation exception, a code meaning "Not-a-Number" or NaN, is not always the best result for interval arithmetic operations. A straightforward implementation of an interval operation can produce an incorrect result if one of its constituent floating point operations incurs such an exception and the computation continues using the default NaN result. Consider, for example, the following algorithm for computing the interval product $[a, b] \times [c, d]$:

$l_1 := a \times c, \quad l_2 := a \times d, \quad l_3 := b \times c, \quad l_4 := b \times d$ rounding down
$l_{12} :=$ if $l_1 < l_2$ then l_1 else l_2
$l_{34} :=$ if $l_3 < l_4$ then l_3 else l_4
$l :=$ if $l_{12} < l_{34}$ then l_{12} else l_{34}
$u_1 := a \times c, \quad u_2 := a \times d, \quad u_3 := b \times c, \quad u_4 := b \times d$ rounding up
$u_{12} :=$ if $u_1 > u_2$ then u_1 else u_2
$u_{34} :=$ if $u_3 > u_4$ then u_3 else u_4
$u :=$ if $u_{12} > u_{34}$ then u_{12} else u_{34}
return $[l, u]$

For the product $[0, 1] \times [1, \infty]$, the preceding algorithm first computes $l_1 = 0$, $l_3 = 1$, and $l_4 = \infty$, but $l_2 = 0 \times \infty = \text{NaN}$. Since the predicate $0 < \text{NaN}$ is false, the algorithm sets $l_{12} = l_2 = \text{NaN}$. It then sets $l_{34} = 1$, and since $\text{NaN} < 1$ is also false, it sets $l = l_{34} = 1$. Likewise, the algorithm computes $u_1 = 0$, $u_2 = \text{NaN}$, $u_3 = 1$, $u_4 = \infty$, then sets $u_{12} = \text{NaN}$, $u_{34} = \infty$, and finally $u = \infty$. Thus the output interval is $[1, \infty]$, which is clearly incorrect: the smallest interval that encloses the product of $[0, 1]$ and $[1, \infty]$ is $[0, \infty]$. (This is precisely the flaw that causes the VerGO software to deliver an incorrect result in the example cited above.)

Of course, most systems that conform to the IEEE standard support its optional trapping mode, so one could simply trap and abort the computation whenever an

invalid operation exception occurs. This practice would be safe but not efficient, forcing programmers to use defensive tests to prevent exceptions rather than simply detecting their occurrence after the fact. Moreover, one of the primary features of interval arithmetic is its ability to bound roundoff errors, which are artifacts of the nonstop mode of handling inexact exceptions. Interval arithmetic can cope with the larger-than-usual roundoff errors resulting from underflows and overflows, too. Since these exceptions can be handled so easily, we might like to handle invalid operation exceptions more gracefully than by aborting.

In this note, we propose an alternative that extends nonstop handling to all IEEE exceptions in real interval arithmetic. We define an interpretation of the real intervals represented by pairs of IEEE floating point numbers and give specifications for a closed, consistent interval arithmetic. In this arithmetic, the obvious algorithm for interval addition always delivers correct results when exceptions are handled according to the IEEE default. Multiplication, division, and square root, however, require some care; we give efficient algorithms for these operations that yield correct results even in the presence of invalid operation exceptions. Finally, we consider some implications of our interpretation and algorithms for floating point hardware support for interval arithmetic.

2 Real Interval Arithmetic in IEEE Floating Point

In order to deliver correct results without forcing program termination in exceptional cases, an implementation of real interval arithmetic must be closed for all operations with operands in some set of representable intervals. Also, the results of exceptional operations must be defined consistently to ensure that evaluating any expression will produce correct bounds. Of course, we would also like to obtain sharp bounds whenever possible, and we would like the implementation to be efficient, requiring a minimal amount of logic to detect and handle special cases. Taking all of these goals into account, we define a set of representable intervals and the results of operations on those intervals.

Definition of Real Interval Arithmetic

Table 1 lists the real intervals we identify with pairs of IEEE floating point values. Naturally, the representable intervals include all intervals of the form $[x, y]$ where x and y are finite floating point numbers and $x \leq y$. To close the arithmetic for operations whose results must contain numbers larger than the largest finite floating point number, we also include intervals unbounded at one or both ends, which we represent using the IEEE floating point numbers $\pm\infty$ as endpoints. Note that we

Representation	Interval	Representation	Interval
$[-0,+0]$	$\{0\}$	$[x,+\infty]$	$\{z : x \le z\}$
$[x,y],\ x \le y$	$\{z : x \le z \le y\}$	$[-\infty,y]$	$\{z : z \le y\}$
$[x,-0],\ x < 0$	$\{z : x \le z < 0\}$	$[-\infty,-0]$	$\{z : z < 0\}$
$[x,+0],\ x < 0$	$\{z : x \le z \le 0\}$	$[-\infty,+0]$	$\{z : z \le 0\}$
$[-0,y],\ 0 < y$	$\{z : 0 \le z \le y\}$	$[-0,+\infty]$	$\{z : 0 \le z\}$
$[+0,y],\ 0 < y$	$\{z : 0 < z \le y\}$	$[+0,+\infty]$	$\{z : 0 < z\}$
$[\mathrm{NaN},\mathrm{NaN}]$	$\emptyset$	$[-\infty,+\infty]$	$\mathbf{R}$

Table 1: Valid floating point intervals and the real intervals they represent. Here x and y may be any finite, nonzero floating point numbers.

do not treat the symbols $\pm\infty$ as real numbers, but only as a means to represent infinite intervals. For this reason, we can define arithmetic on intervals with infinite endpoints without encountering dilemmas over the definition of arithmetic on the "number" infinity; such dilemmas are the source of several of the invalid operation exceptions in IEEE floating point. For example, because we do not admit the pairs $[-\infty,-\infty]$ and $[+\infty,+\infty]$, we avoid the $\infty-\infty$ invalid operation exception in interval addition.

One invalid operation that poses a problem for real interval arithmetic is the square root of a negative interval. To close the arithmetic for this operation without introducing complex intervals, we must deliver a result that indicates that square root does not take real values on the negative real axis. The IEEE special value NaN is a convenient way to represent such a result, particularly since the standard requires that the square root of a negative number deliver NaN. Therefore, we propose to deliver the result $[\mathrm{NaN},\mathrm{NaN}]$ for the square root of a strictly negative interval.

This choice raises a question: how should we handle the square root of an interval that contains both negative and nonnegative numbers? Simply taking the square root of each endpoint would produce an interval with NaN at only one end. As Popova[10] observes, such intervals can be difficult to handle correctly in interval operations such as multiplication and division, which typically branch on comparisons of floating point numbers. (The reason the BIAS software delivers an incorrect result in the example in section 1 is that its implementation of interval multiplication can not only produce a NaN result from real interval operands but also deliver a real result when its operands involve NaNs.) Instead, Popova suggests restricting the interval argument of a function to the real-valued domain of that function, so that $\mathrm{sqrt}(X)$ is implicitly interpreted as $\mathrm{sqrt}(X \cap [0,\infty))$. Of course, if X is strictly negative, the intersection is empty. Because we deliver the result $[\mathrm{NaN},\mathrm{NaN}]$ in that case, it seems natural

to interpret [NaN, NaN] as a representation of the empty set. Moreover, unlike an interval with NaN at only one end, the interval [NaN, NaN] does not pose any special difficulty for the implementation of interval arithmetic operations: if either operand is [NaN, NaN], the result is [NaN, NaN] also. Since the result of any operation with an empty operand should be empty, this is another reason to identify [NaN, NaN] with the empty set.

Most of the remaining invalid operations in IEEE floating point are related to the indeterminate form $0/0$. (Its relatives are $0 \times \infty$ and ∞/∞.) In real interval arithmetic, there is often only one reasonable result we can deliver for a quotient A/B when B contains zero: for example, if A contains any nonzero number and B contains zero in its interior, we must deliver $[-\infty, +\infty]$, and if A does not contain zero and B is a degenerate interval at zero, we should deliver [NaN, NaN], since there is no real number x such that $a = xb$ for some $a \in A$ and $b \in B$. By analogy with this last argument, one might be tempted to define $A/B := [-\infty, +\infty]$ whenever B is a degenerate interval at zero and A contains zero. This would be tantamount to filling in each removable singularity in a rational function with a vertical line. Unfortunately, removable singularities do not always appear in such a simple form; even the obvious transformation $A/B = A \times (1/B)$ turns a quotient that one might want to be $[-\infty, +\infty]$ into a product that should be empty. When we consider other transformations such as $(A/B)^2 = (A^2/B)/B = A \times (A \times (1/B)^2)$, we find that it is not easy to define division by a degenerate interval at zero consistently in this way. We therefore propose a different approach: following the example of square root, we implicitly interpret the quotient A/B as $A/(B \setminus \{0\})$. This interpretation simply leaves removable singularities empty.

We can summarize the preceding observations in two simple specifications for an implementation of real interval arithmetic. For brevity, we first introduce a definition.

DEFINITION: Call an ordered pair of floating point values a *valid* floating point interval if it is one of the representations shown in table 1.

Evidently an implementation of real interval arithmetic will deliver correct results even in the presence of exceptions if it meets the following specifications:

1. The computed sum, product, or quotient of two valid intervals is valid, and the computed square root of a valid interval is valid.

2. The computed sum or product of intervals A and B contains the exact sum or product of those intervals. The computed quotient A/B contains the set $\{a/b : a \in A, b \in B, b \neq 0\}$. The computed square root sqrt(A) contains the set $\{\sqrt{a} : a \in A, a \geq 0\}$.

In particular, the quotient of any interval divided by a degenerate interval at zero

and the square root of a strictly negative interval may be empty.

Algorithms for Real Interval Arithmetic Operations

We have defined the representable real intervals and specified the results of arithmetic operations on them to facilitate efficient implementations of these operations in IEEE floating point. Our implementations rely for their efficiency on the default results and exception flags required by the IEEE standard. For example, as noted above, the obvious algorithm for interval addition meets both of the preceding specifications because we have defined the representable intervals so that invalid operation exceptions cannot arise; the only other exception besides inexact that can occur in interval addition is overflow, for which the IEEE default results give correct bounds. In this section, we give algorithms for interval multiplication, division, and square root that also meet our specifications.

Our algorithms also rely on the convention that a nonempty interval contains zero if and only if its endpoints have opposite signs. Of course, this is obviously true of intervals with nonzero endpoints. To extend this convention to intervals with zero endpoints, we adopt an idea suggested by Kahan[5] to use the sign bit in the floating point representation of zero to distinguish open and closed interval endpoints at zero. For example, the interval $[-0, +1]$ includes zero at its left endpoint, but $[+0, +1]$ does not. Likewise, $[-0, -0]$ and $[+0, +0]$ are not valid representations of a degenerate interval at zero; only $[-0, +0]$ is. Fortunately, this convention was designed into the IEEE standard's rules prescribing the sign of a zero result, so interval addition automatically follows the convention. In interval multiplication, we must be careful to ensure that zero endpoints of products have the correct sign, but as the following algorithm shows, this requirement is easy to satisfy in a software implementation. As we will show in section 3, with suitable hardware support, we can satisfy it at no extra cost.

The following algorithm computes the interval product $[a, b] \times [c, d]$ and meets both specifications given above. The algorithm selects one of nine cases based on the signs of the endpoints of the factors to determine which endpoints will give the extreme bounds for the interval product. This method can encounter an invalid operation exception when one of the endpoint products has the form $0 \times \infty$; in that case, the correct result is obtained by replacing the product by a correctly signed zero. Here, the integer variable s keeps track of the appropriate signs. (s is 0 if the interval product is strictly positive, 1 if it is strictly negative, and 2 if it contains zero.) Note that the tests $y \not= y$, etc., are used to check for NaNs: in IEEE arithmetic, the predicate $x \not= x$ is true precisely when x is NaN. Also, when either l or u is found to be NaN, we use the tests $a \leq b$ and $c \leq d$ to distinguish a NaN arising from a $0 \times \infty$ product from one propagated from an empty operand. (To avoid spurious

exceptions, these tests as well as the tests $x < y$ and $z > w$ in the ninth case should be implemented with predicates that do not raise an invalid operation exception when either operand is a quiet NaN.)

```
if signbit(a) = 0
    if signbit(c) = 0
        l := a × c rounding down
        u := b × d rounding up
        s := 0
    else if signbit(d) = 1
        l := b × c rounding down
        u := a × d rounding up
        s := 1
    else
        l := b × c rounding down
        u := b × d rounding up
        s := 2
else if signbit(b) = 1
    if signbit(c) = 0
        l := a × d rounding down
        u := b × c rounding up
        s := 1
    else if signbit(d) = 1
        l := b × d rounding down
        u := a × c rounding up
        s := 0
    else
        l := a × d rounding down
        u := a × c rounding up
        s := 2
else
    s := 2
    if signbit(c) = 0
        l := a × d rounding down
        u := b × d rounding up
    else if signbit(d) = 1
        l := b × c rounding down
        u := a × c rounding up
    else
        x := a × d rounding down
        y := b × c rounding down
        l := if (x < y or y ≠ y) then x else y
```

```
        z := a × c rounding up
        w := b × d rounding up
        u := if (z > w or w ≠ w) then z else w
if l ≠ l or u ≠ u
    if a ≤ b and c ≤ d
        if l ≠ l
            l := if s = 0 then +0 else −0
        if u ≠ u
            u := if s = 1 then −0 else +0
return [l, u]
```

We have defined interval division by excluding zero from the divisor. One consequence of this definition is that we can implement interval division as multiplication by a reciprocal. In particular, the following algorithm computes the interval quotient $[a, b]/[c, d]$ and meets both of our specifications. Note that if the divisor is a degenerate interval at zero, we explicitly divide its endpoints both to obtain the correct result $[\mathrm{NaN}, \mathrm{NaN}]$ and to raise the invalid operation exception flag. Similarly, if the divisor contains zero, we explicitly divide by zero both to obtain infinite endpoints in the quotient and to raise the division-by-zero exception flag. By raising these flags in addition to delivering the default results, we allow a user's program to detect these exceptions after the fact. (To avoid spurious exceptions, the test $c < d$ in the fifth line should be implemented with a predicate that does not raise an invalid operation exception if either operand is a quiet NaN.)

```
x := d,    y := c
if signbit(c) ≠ signbit(d)
    if c = d
        return [c/d, c/d]
    else if c < d
        if c < 0
            x := −0
        if d > 0
            y := +0
z := 1/x rounding down
w := 1/y rounding up
[l, u] := [a, b] × [z, w]
return [l, u]
```

Finally, the following algorithm computes the interval square root $\mathrm{sqrt}([a, b])$ and meets both of our specifications. Note that we compute the square root of each endpoint of the argument before testing whether it contains negative numbers. This ensures that we raise the invalid operation flag even when the argument contains both

negative and nonnegative numbers, so that we provide an indication that something unusual happened even though the result appears unexceptional. (To avoid spurious exceptions, the test $a \le b$ in the third line should be implemented with a predicate that does not raise an invalid operation exception if either operand is a quiet NaN.)

```
l := sqrt(a) rounding down
u := sqrt(b) rounding up
if a ≤ b
    if signbit(b) = 1
        u := l
    else if signbit(a) = 1
        l := −0
return [l, u]
```

3 Hardware Support

The preceding algorithms might suggest that implementing interval arithmetic in IEEE floating point requires changing the rounding direction frequently. This could be a performance bottleneck, since changing the rounding direction is an expensive operation on many systems. Fortunately, we need not change rounding directions often. Except for the square root operation, IEEE arithmetic in the round-to-negative-infinity mode is completely symmetric with arithmetic in the round-to-positive-infinity mode. Thus, rather than represent an interval in storage by its upper and lower bounds, we can instead store its lower bound and the negative of its upper bound, or vice versa; we can then compute both bounds in the same rounding mode[1]. This technique saves numerous rounding mode changes at a cost of one unary negation for each input and output and at most two negations for each multiplication and division. Addition incurs no extra cost. With care, we can even compute both endpoints of an interval square root in the same rounding direction: for example, in round-to-positive-infinity mode, the code fragment

```
l := −sqrt(−x)
if l × l > −x
    l := l × nextafter(1, 0)
```

delivers the negative of the nearest representable lower bound to $\sqrt{-x}$.[1] In principle, then, we only need to change the rounding mode once to begin a sequence of interval operations and again to return to point operations.

[1]This implementation, however, can raise a spurious underflow exception when x is tiny and a spurious overflow exception when x is the largest finite floating point number. A similar technique for computing the negative of the upper bound in round-to-negative-infinity mode avoids spurious overflow but not spurious underflow. At the cost of a floating point division and an extra rounding

Even if we can avoid changing rounding modes, an implementation of interval multiplication based on the nine-case method will be less than optimal on modern, heavily pipelined superscalar processors. On most such processors, testing the sign bit of a floating point number must be done using integer operations, thereby requiring that data in floating point registers be stored and loaded into integer registers. (There are a few exceptions; for example, on UltraSPARC systems, one can use the VIS instructions to test the sign bit of a floating point register[11].) Moreover, the tests and branches of the nine-case method tend to prevent hardware and compilers from cooperating to fully utilize the floating point pipeline to exploit instruction-level parallelism.

Hough[4] suggests that with hardware support for branchless min and max operations, an implementation based on the usual definition

$$[a, b] \times [c, d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)]$$

would be more efficient. The eight products could be computed in parallel, and some of the latency in the min and max reductions could be hidden by arranging them in binary tree form. Most important, because the implementation would not require any branches, an optimizing compiler could freely schedule other computational instructions among the multiplications and reductions. Such an implementation would require special consideration to handle the $0 \times \infty$ invalid operation exception as well as NaNs and signed zeroes, however. For example, because -0 and $+0$ compare equal, implementations of min and max using IEEE comparisons cannot easily support the convention that the endpoints of a product have opposite signs whenever the product contains zero. Likewise, although we can arrange the reductions in interval multiplication so that a NaN in either factor will propagate to the product, in more general contexts, min and max based on IEEE comparisons cannot easily handle NaNs correctly, either. Thus, to handle NaNs and signed zeroes efficiently, we need hardware primitives that support efficient implementations of min and max satisfying

(i) $\min(x, y) = \min(y, x)$ for all x, y, and similarly for max,

(ii) $\min(x, \mathrm{NaN}) = \max(x, \mathrm{NaN}) = \mathrm{NaN}$ for any x, and

(iii) $\min(-0, +0) = -0$, $\max(-0, +0) = +0$.

To handle $0 \times \infty$ invalid operations in interval multiplication, we want to deliver an appropriately signed zero for the result of each $0 \times \infty$ product. This could be achieved

---

error in the result, the assignment

$l :=$ if $x = 0$ then x else $x/\mathrm{sqrt}(-x)$

in round-to-positive-infinity mode delivers the negative of a lower bound for $\sqrt{-x}$ with no spurious exceptions.

using a feature Kahan[6] calls "presubstitution". Presubstitution extends the IEEE default response to each exception by allowing the user to specify the value to deliver when that exception occurs. In one form of presubstitution, the substituted result has the magnitude of the specified value with the sign of what would have been the default result. For our purpose, the sign of the substituted result must be inferred from the operands, since the sign of the default NaN delivered for an invalid operation exception is meaningless. Presubstitution can be implemented on current systems via a trap handler, but such implementations are awkward and inefficient. Alternatively, presubstitution could be supported in hardware by adding a few special purpose registers to hold presubstitution values, but such a general mechanism would require software coordination similar to that needed to cope with rounding and trapping modes. For interval multiplication, we prefer a simpler approach: provide an alternate floating point multiply operation that delivers zero rather than NaN for $0 \times \infty$.

More complete hardware support for interval arithmetic would combine features such as those we have described with the ability to switch rounding direction quickly so that we can handle both special and general cases as efficiently as possible. One approach to changing rounding modes quickly, which has been implemented in the Cray T90[8], copies the rounding mode bits from the floating point control register and propagates them through the execution pipeline with each instruction, so that the bits in the control register may be changed to affect new instructions without waiting for older instructions to complete. With special instructions that update only the rounding mode bits leaving the other control and status bits unchanged, we can avoid saving and restoring the entire control register on each mode change. This approach is perhaps the most attractive way to adapt existing implementations of IEEE floating point to support fast interval arithmetic, since it requires only modest changes to hardware and the addition of a few extra instructions. On the other hand, this method still requires explicit instructions to change rounding modes, and optimizing compilers must be programmed to respect the interaction between those instructions and the floating point operations they affect. Moreover, the presence of a mode of any kind implies the need to save and restore information when switching between different tasks either within a program or from one program to another, and this bookkeeping carries some cost.

The best way to provide directed roundings for interval arithmetic is to bypass the IEEE standard's rounding modes and encode rounding direction directly in each floating point opcode. (DEC's Alpha architecture[3] does this in an unfortunately awkward way: two bits in each opcode select one of three hard-wired rounding directions or the direction specified by the ambient rounding mode, but round-to-negative-infinity is not one of the three hard-wired directions.) With this approach, interval arithmetic could be implemented using instructions with the rounding directions hard-wired, eliminating both the need to emit extra instructions to change the

rounding direction and the need to save and restore modes. Moreover, multiplication instructions with hard-wired rounding directions could support the special treatment of $0 \times \infty$ we have described above, leaving the standard mode-based multiply to provide IEEE behavior. Even though this method requires a large number of additional floating point opcodes and the attendant hardware needed to decode them, we believe that it will prove superior for high-performance implementations of interval arithmetic.

4 Concluding Remarks

We have proposed an interpretation of real intervals represented by certain pairs of IEEE floating point values and shown that we can realize a closed, consistent arithmetic on these intervals by efficient algorithms that deliver correct results even when exceptions occur. We have also suggested a way to enhance floating point hardware to allow a straightforward implementation of this arithmetic to cope with exceptional cases at no extra cost.

Our proposals are by no means complete. For example, an implementation must also handle exceptions in interval comparisons and non-arithmetic operations such as hull and intersection; as Popova[10] shows, these operations must be implemented carefully to deal with NaNs. Likewise, an implementation must decide what result to deliver when the argument of an intrinsic elementary function lies partially or completely outside the real-valued domain of that function. By analogy with the square root operation, it seems desirable to intersect the argument with the real-valued domain, but no matter what decision is made, the implementation should raise an exception flag to let the user know that something suspicious happened. Finally, in the interest of efficiency, we have restricted our interpretation to real intervals and imposed a particularly simple rule to define division by intervals containing zero. Kahan[5] proposes a more complete extended arithmetic based on connected subsets of one-dimensional real projective space. His extension includes "exterior intervals" that preserve more information in a quotient with a divisor containing zero at the cost of requiring much more logic to interpret the representation. Thus, our proposals reflect choices and tradeoffs typical of those we must make to develop better tools for numerical computation.

The ideas we have proposed represent an attempt to enhance the reliability and utility of interval arithmetic by incorporating an interval analogue of the IEEE nonstop exception handling paradigm. The ability to handle floating point exceptions gracefully, in particular to substitute reasonable default results for exceptional operations, is one of the most useful (if least appreciated) features of IEEE floating point; when used properly, it allows users to ignore most exceptions most of the time and to de-

tect and handle the rest after the fact. As we have shown, we can turn this feature to our advantage in interval arithmetic, in some cases delivering potentially better results than the IEEE default NaN in the presence of invalid operation exceptions. By paying attention to floating point exceptions, making careful choices about how best to handle them, and considering the impact of our choices on future hardware designs, we can make interval arithmetic both safe and fast.

Acknowledgements

I would like to thank G. W. Walster for providing helpful comments on a preliminary draft of this paper.

References

[1] Alverson, R., post to `numeric-interest@validgh.com` mailing list, 1995.

[2] ANSI/IEEE 754-1985 Standard for Binary Floating-Point Arithmetic, Institute of Electrical and Electronics Engineers, New York, 1985.

[3] Digital Equipment Corporation, *Alpha Architecture Handbook*, 1992.

[4] Hough, D., post to `numeric-interest@validgh.com` mailing list, 1995.

[5] Kahan, W., A More Complete Interval Arithmetic, lecture notes prepared for a summer course at the University of Michigan, 1968.

[6] ——, Lecture Notes on the Status of IEEE Standard 754 for Binary Floating-Point Arithmetic, unpublished notes, 1996. Available from

`http://http.cs.berkeley.edu/~wkahan/ieee754status/ieee754.ps`

[7] Kearfott, R. B., M. Dawande, K. Du, and Ch. Hu, Algorithm 737: INTLIB: A Portable Fortran-77 Elementary Function Library, *ACM Trans. Math. Soft.* **20** (1993), 447–459.

[8] Kiernan, J., and W. Harrod, Implementation of IEEE floating-point arithmetic on the Cray T90 system, *Cray Users' Group Fall Proceedings*, 1995.

[9] Knüppel, O., PROFIL/BIAS—A Fast Interval Library, *Computing* **53** (1996), 277–287.

[10] Popova, E., Interval Operations Involving NaNs, in G. Alefeld and A. Frommer, Eds., *Scientific Computing and Validated Numerics: Proceedings of SCAN-95*, Akademie-Verlag, Berlin, 1996.

[11] Sun Microsystems, Inc., *UltraSPARC User's Manual*, Revision 2.0, 1996.

[12] Van Iwaarden, R., Global Optimization using VerGO: Verified Global Optimization in C++, 1996. Software available from

`http://www.cs.hope.edu/~rvaniwaa/VerGO/VerGO.html`