The Minefield Method: A Uniformly Fast Solution to the Table-Maker's Dilemma

John L. Gustafson
School of Computing
National University of Singapore
Singapore
john.gustafson@nus.edu.sg

***Abstract*—In computing transcendental functions, some function values can be so close to the tie point between rounding up or rounding down that ultra-high-precision methods are needed to determine the correct rounding. This "Table-Maker's Dilemma" has led math library designers to allow a few results to round incorrectly in the interest of performance, at the expense of creating irreproducibility across computing systems. Another approach is to use a fast method for the majority of results that have unambiguous rounding, but trap to a more expensive method when the fast method cannot resolve the tie; however, this produces slower average speed, unpredictable speed, and a security hole via the side-channel attack of timing the computations. The *minefield method* presented here performs perfect rounding without expensive methods such as high-degree approximations and extended-precision arithmetic. Routines with perfect rounding have been built for logarithm, exponential, trigonometric and inverse trigonometric functions that require only about 30 clock cycles per result on a modern microprocessor, and the worst-case timing is scarcely different from the average timing. This result is made possible by the insight that it is *not* necessary to minimize the maximum error to the worst case accuracy needed; it is only necessary to insure that the error is *always in the safe direction*. The minefield approach also shows how the long-standing problem of provably correct rounding for the power function can be made tractable; we observe that machine-representable inputs to the power function are *rational* numbers, which means the function is *algebraic*, not *transcendental*.**

***Index Terms*—computer arithmetic, error analysis, numerical analysis, numerical algorithms and problems, mathematics of computing**

I. Introduction

Since the first logarithm tables were created hundreds of years ago, table-makers noticed that certain table values were extraordinarily (and unpredictably) difficult to round correctly. For example, if the table-maker seeks six decimals of accuracy and the algorithm finds that the function is somewhere between $0.7231234999\cdots$ and $0.7231235000\cdots$ (where "$\cdots$" indicates there are more unknown nonzero digits), the calculation of three additional decimal digits is not sufficient to determine if the correct rounding to six decimals is to 0.723123 or 0.723124.

The defense for the "it does not matter" position is that both rounding choices are almost equally close to the mathematical value, hence we quibble if we demand correct rounding. It seems unlikely that a practical calculation would be spoiled by picking the wrong rounding direction. However, the issue of *reproducibility* offers a powerful argument for "it matters": Just as we would never accept a computing environment that returns $2 + 3 = 6$ using some libraries and $2 + 3 = 5$ using other libraries, why should we accept a computing argument where the values of $\exp(x)$, $\cos(x)$ and so on vary in the last bit depending on the source of the software for evaluating those functions? Inconsistency undermines trust in computing systems, for example, when a financial model is run on a large portfolio and its multibillion-dollar value varies by tens of thousands of dollars depending on which standard math library is used. It creates suspicions: Is there a bug? Was there a soft error? Is the hardware defective?

If inconsistency in math library functions can be eliminated without performance degradation, there would be one fewer issue that programmers must cope with. Perfect rounding eliminates inconsistency. It also preserves the monotonicity of the function being approximated, which is important for some numerical methods. Can perfection be made consistently *fast*? The extensive work of Lefèvre and others to discover and exhaustively tabulate hard-to-round values [**?**] has provided a crucial step toward a solution to this problem.

A. The IEEE 754 position and currently-available accuracy

The 2019 version of the IEEE 754 Standard for Floating-Point Arithmetic *recommends*, but does not *require*, correct rounding for a specific set of mathematical functions [**?**], [**?**]. Correctly-rounded open-source math libraries have been produced by IBM (APMathLib) [**?**], Sun (libmcr) [**?**], and the Arènaire team (CRlibm) [**?**]. The mainstream libraries, both open-source and commercial, do not adhere to this level of perfection; Some values will be incorrectly rounded, and therefore off by one Unit in the Last Place (ULP). Most values will be correctly rounded, so the average error might be something like 0.501 ULPs. An average error of exactly 0.5 ULPs means perfect rounding, but might cost orders of magnitude more work to achieve for some input arguments. The classic methods of Cody and Waite generally aim for accuracy within a few ULPs [**?**] and have been improved by Beebe but still cannot claim perfect rounding [**?**].

B. The Emerging Posit Standard Position

Posit arithmetic has been proposed as a replacement for IEEE 754 floating-point arithmetic, and has attracted some

interest. In the Draft Posit Standard [**?**], the requirement for correctly-rounded math functions, including transcendental functions, is unequivocal. Kahan has stated that bitwise reproducibility is an impractical goal, largely because of the Table-Maker's Dilemma, a term that he coined [**?**]. Therefore, an open source and uniformly fast method of evaluating the transcendental functions with correct rounding must be found to meet this bold requirement, and this is the motivation for the present work.

II. The Fallacy of the Minimax Argument

An example of the traditional view of the Table-Maker's Dilemma is shown in Fig. **??**. This example uses half-precision IEEE floats to avoid values with long decimals, but the Dilemma occurs at any precision. The gray dots are representable values spaced one ULP apart in both the domain and the range. The *tie points* are the hollow circles halfway between each vertical pair of gray dots. For the middle point, the tie point lies so close to the exact function value that it is hard to decide which way to round.

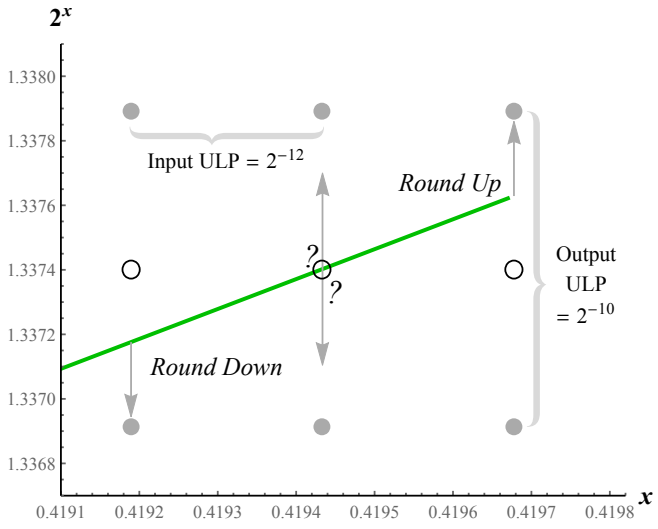

Fig. 1. Example of the Table-Maker's Dilemma, traditional view.

Suppose we calculate the function to a fraction of a Unit in the Last Place (ULP), say, six bits beyond the needed significance. That accuracy will be sufficient to resolve that the result for the left input value should round down, and the result for the right input value should round up. For the center point, however, an extraordinarily accurate calculation is needed to decide which way it should round. The input x value is $\frac{1718}{4096}$, and $2^{\frac{1718}{4096}}$ in binary is `1.0101011001`100000$\cdots$ where the gray bits determine rounding. Even with six extra bits of accuracy, we do not know whether it should round up or down. For this input value, the function has to be evaluated to *25* significant bits (more than twice the significand size) to discover it should be rounded up: $2^{\frac{1718}{4096}}$ = `1.0101011001`10000000000001$\cdots$. In the scale of fig **??**, the line passes above the exact tie-breaking point by only *two microns*. For single- and double-precision standard floats, the problem can be astronomical, with hundreds of extra bits needed to resolve the rounding for a few of the results.

A. Current approaches to dealing with the Dilemma

A numerical approximation that minimizes the maximum error, such as a polynomial or rational function, and has such a small error that all rounding directions are resolved correctly, is generally so expensive that both the creators of math libraries and the computer users who choose which libraries to link will usually opt for methods short of such perfection [**?**].

Kahan writes: "Linguistically legislated exact reproducibility is unenforceable [**?**]."

A somewhat more palatable approach is to use a more modest number of extra bits beyond the working precision, say ten bits, but trap the cases where the bits to be rounded match the difficult cases of `1000000000` or `0111111111`. Statistically, those bit patterns would arise about two in every 1024 cases, and double rounding will produce the wrong rounding about half the time. The trap can then invoke successively higher-precision methods so that the very slow cases only happen rarely. Zif was the first to suggest this "peeling the onion" approach [**?**], [**?**]. There are several drawbacks to this approach:

- Application performance becomes highly unpredictable, possibly surprising the user by several orders of magnitude slowdown in the worst case.
- The average performance will be well below that of a function evaluation that tolerates a few incorrectly-rounded results.
- Data-dependent timings open up a side-channel security attack [**?**].

Interval computing environments also experience a form of the Table-Maker's Dilemma because they need to declare a pair of numbers one ULP apart that describe the interval guaranteed to contain the result of a transcendental function evaluation. Instead of a result being very close to the tie point, the challenge is when a result is very close to an expressible real number used as an endpoint. Is the enclosing interval above or below the computed value, in that case? A simple solution is to let the answer be two ULPs wide, in effect admitting that the bound is not tight.

The most common way of dealing with the Table-Maker's Dilemma is that used by Intel, AMD, Gnu, and other suppliers of optimized math libraries for standard languages: Admit defeat. Accept a few incorrectly-rounded results, even when it creates different results on different computing systems. This follows Kahan's position that insistence on bitwise-reproducibility is an impractical goal.

In the next Section, we present a new way of viewing the Table-Maker's Dilemma as a "minefield" plot, which gives rise to a way to achieve perfect rounding that is **uniformly fast**, without using tables or high-precision arithmetic libraries or very high-order approximations. While it does require manual tuning for each function (for now), the procedure is tractable

in that it does not require exhaustive testing for all possible input arguments, only for the hard-to-round cases.

III. The Minefield Method

Instead of Fig. **??**, consider all the *tie points* above and below the transcendental function $y = f(x)$ we wish to evaluate. We are careful to distinguish several versions of y.

y^* is a calculated approximation to y using some method.
$\bar{y}$ is the correctly-rounded value of y.
$\bar{y}^-$ is the value one ULP less than $\bar{y}$.
$\bar{y}^+$ is the value one ULP greater than $\bar{y}$.
y_i is $f(x_i)$ where $x_1, \ldots, x_n$ are representable input values.

We have perfect rounding on the domain of x_i values when $y_i^* = \bar{y}_i$ for all i. By round-to-nearest rules, this means y_i^* lies between the tie points below and above $\bar{y}$:

$$\tfrac{1}{2}(\bar{y}_i + \bar{y}_i^-) < y^* < \tfrac{1}{2}(\bar{y}_i + \bar{y}_i^+) \tag{1}$$

The coordinate pair $(x_i, \frac{1}{2}(\bar{y}_i + \bar{y}_i^-) - y_i)$ is a *low mine* because if the plotted curve of the approximation to $f(x) - y$ falls below that point, it will round incorrectly and destroy any attempt at perfect rounding. Similarly, the coordinate pair $(x_i, (\bar{y}_i + \bar{y}_i^+)/2 - y)$ is a *high mine*. The *minefield* is the set of a low mines and high mines for the domain x_1 to x_n.

A. Example for a reduced argument

Algorithms for transcendental functions typically start by handling exception regions where the function is trivial to evaluate (outside the domain, overflows, always rounds to 1, etc.). Then argument reduction is used to restrict the values being approximated to a manageable small range. Exponential functions a^x can be evaluated this way. Fig. **??** shows the minefield for the function $f(x) = 2^x, 1 \leqslant x \leqslant 2$, where the number format has an ULP spacing of 2^{-13} in this domain.

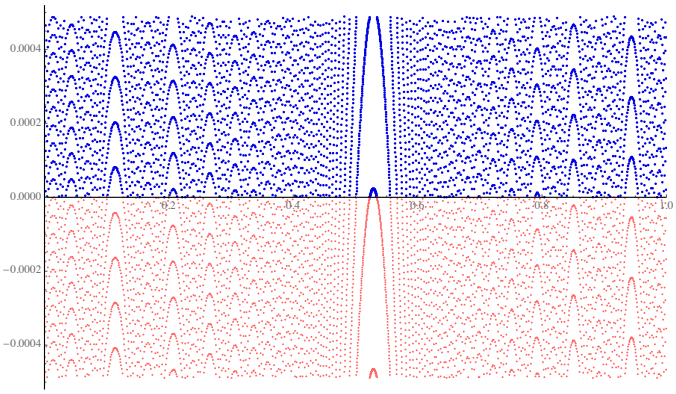

Fig. 2. Minefield for the 2^x function, 13-bit significand

The pattern is not random, but fractal-like. The height of the full minefield is one ULP, but as we saw previously, the Table-Maker's Dilemma occurs at a much smaller scale than one ULP. Fig. **??** expands the vertical axis to show the mines we need to pay attention to.

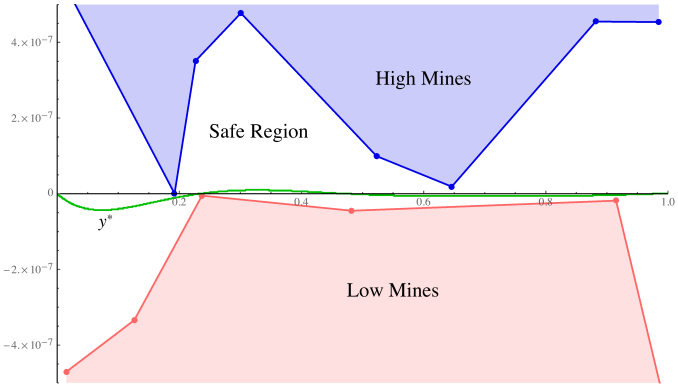

Fig. 3. Minefield for the 2^x function, magnified.

The x-axis now represents the exact value of $f(x)$, and we can see how daunting it is to find a minimax approximation that hugs the x-axis so closely that it never touches any of the mines. But we can also see from this new way of looking at the problem that **it is not necessary to be everywhere close** to the x-axis; we need only stay in the envelope between the low mines and the high mines. A relatively low-order approximation suffices to steer through the minefield, By making correct rounding the *direct goal* instead of hoping for it as the consequence of high accuracy, we discover that perfection can be had at surprisingly low cost.

The approximating function is found manually, not theoretically, by trial-and-error selection of parameters in the approximating function such as polynomial coefficients and the degree of the approximating polynomial. One approach is to specify the roots of the approximation in the minefield domain, and slide their location left and right on the x-axis until the curve never passes above the high mines or below the low mines. This is a major departure from most approaches because it relies on a slower "oracle" function evaluation and then makes use of known hazards found by the oracle, instead of using first principles to guarantee correct rounding. It does not preclude the possibility that the approximating function can be *proved* to lie entirely inside the safe region and thus always round correctly.

A final step is to convert the computation of the approximating function to a fixed-point calculation (integer multiplies followed by shifts) that uses the lowest-possible precision. This makes the sharp curve of the approximating polynomial shown in Fig. **??** into a blurred band of points. The safety of the approximation can be quickly tested by evaluating it only at the minefield points, not the entire set of points in the domain.

B. Using the Minefield on the full function domain

Most library functions use some form of argument reduction. In computing a function like e^x with binary floats, the problem is converted to $2^{x \log_2(e)}$ so that $\lfloor x \log_2(e) \rfloor$ becomes the exponent of the formatted number and the remainder is the reduced argument. For all such methods, it is crucial to note that **the minefield is the union of the minefields of**

each binade. In the case of floats, this can mean combining the minefields of all regions for which e^x does not underflow or overflow before determining an approximating function that avoids hitting any mines. This will be shown more explicitly in Section **??**.

C. The 16-bit posit math library

While the minefield method certainly works for floating-point and fixed-point formats, we have focused on posit arithmetic to see if the insistence in the Draft Standard for perfect rounding is feasible for the transcendental functions. For 16-bit posits, the significand is two bits longer than for IEEE binary16 (13 bits instead of 11 bits), adding slightly to the challenge of making the routines fast. The following perfectly-rounded transcendental functions have been completed and open-sourced at [*name hidden for anonymous review*] :

Trigonometric: `cospi`, `sinpi`, `tanpi`
Exponential: `exp`, `exp2`
Logarithmic: `log`, `log2`
Inverse trigonometric: `acos`, `acospi`, `asin`, `asinpi`, `atan`, `atanpi`

All involve polynomial or rational approximations with fewer than seven terms and corresponding adjustable coefficients. The only timing irregularities are certain very fast values that require little or no arithmetic, such as small input values x for which $\arctan(x)$ rounds to x. In contrast with other correctly-rounded libraries, these routines do not use double or double-double floating-point arithmetic. Instead they use native 64-bit integer arithmetic, which for register-to-register operations is actually faster than 32-bit or 16-bit integer arithmetic on modern x86 processors. Also, while a 64-bit floating-point multiply-add has a latency of seven or eight clock cycles, a 64-bit integer multiply-add now has a latency of only two clock cycles and has more bits of significance than the float operations. Even casual coding of Horner's rule with none of the usual tricks to help superscalar performance produced speeds of about 100 million evaluations per second on a single ~ 3 GHz Intel processor core. In other words, the routines take about 30 clock cycles per result, other than the few exception cases that run much faster. If posit arithmetic is performed with software emulators, addition and subtraction also take about 30 clock cycles; until posit arithmetic becomes native in mainstream processor hardware, we have the surprising result that for posits, functions as complicated as arccosine and logarithm are similar in speed to that of the four elementary functions of arithmetic.

IV. A Complete Function for 32-bit Precision

For higher precision, one might think the large number of cases to test makes perfection intractable. However, because the number of very close mines is small, and because avoiding them also avoids all other rounding errors when the approximating function is smooth, trial-and-error is actually quite easy to do in a graphical environment. We here show details of a routine for computing 2^x for 32-bit posit arithmetic, which is similar to 32-bit float arithmetic but with tapered precision. In the highest-accuracy region of 32-bit posits, there are 28 bits in the significand compared to 24 in the significand for IEEE binary32 [**?**], [**?**].

About 8 percent of all possible input values can be dispensed with quickly, without calculations. Fig. **??** shows the projective real number ring used as the basis for posits, with the point at infinity at the top serving as a catch-all Not-a-Real replacement for the signed infinity and NaN exceptions of floats.

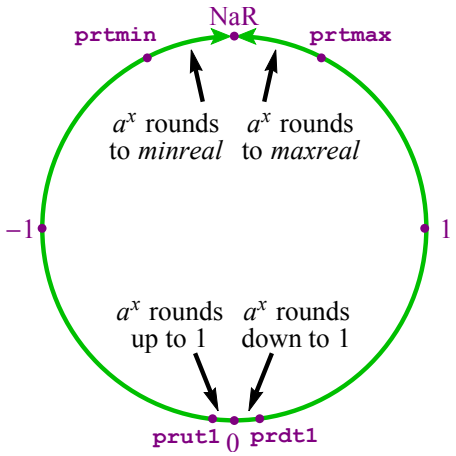

Fig. 4. Exception regions for a^x, projective reals.

While it might seem almost simpler to let values close to 0 be computed with whatever approximation works for the broader region, there are many mines in the minefield between `prut1` and `prdt1` at the bottom of Fig. **??**, and that reduces the number of mines we need the approximation to avoid. Fig. **??** shows the mines for inputs between 0 and 1 with the vertical scale magnified to show just the hard-to-round cases.

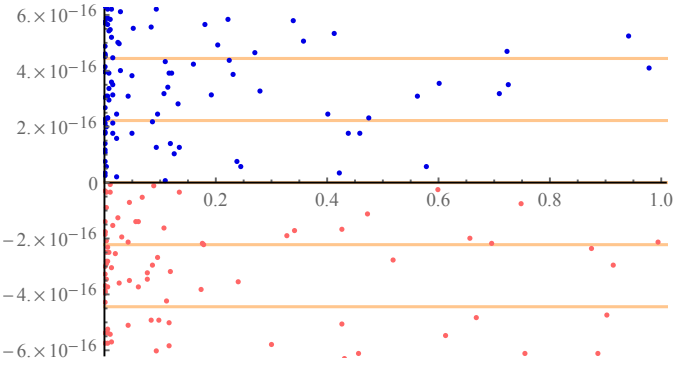

Fig. 5. Magnified view of 2^x minefield for $0 \leqslant x \leqslant 1$

The horizontal lines show the ULP spacing of IEEE 754 *double-precision* values in the range of the function, as a warning to those who would attempt to achieve perfect rounding by evaluating an accurate 64-bit `exp2(x)` and converting

the binary64 result to the nearest 32-bit posit. Because of the Table-Maker's Dilemma and double rounding, that approach will produce incorrect rounding for ten of the possible input values in 0 to 1.

After dealing with the exception regions, we tackle the minefield in three pieces, because the concentration of mines is much higher on the left than the right in the range shown. Up to $x \approx 0.0012$, a cubic polynomial serves to avoid the mines. Then a sixth-degree (hexic) polynomial works well up to $x \approx 0.21$. Finally, a tenth-degree (decic) polynomial suffices to avoid the mines between about 0.21 and 1. We sample the interval at a spacing of 2^{-52} to cover all the uses the approximation with reduced arguments.

A. The cubic approximation

Fig. **??** shows the result of selecting roots of a cubic and plotting it interactively along with the mines, adjusting the roots so the approximation works for as large a domain as possible without triggering any mines. The rightmost mine, near $x = 0.0012$, shows why a different function is needed beyond that input value. The approximation clearly goes below that low mine and will cause an incorrect rounding.

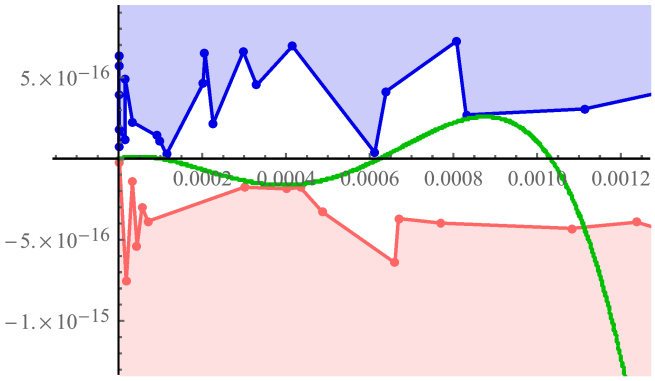

Fig. 6. Mine-avoiding cubic polynomial

While it may seem like 0 to 0.0012 is a tiny region, the cubic polynomial actually covers over 200 million points, about 20 percent of the number of input values between 0 and 1. When converted to all-integer multiplies, shifts, and adds, the function becomes fuzzier, as shown in Fig. **??**. Two of the multiplies are slightly larger than 64 bits and must be done in two pieces, so the total cost of the cubic is 5 multiplies, 4 adds, 2 ANDs, and 7 right shifts; a superscalar architecture schedules many of those instructions to execute concurrently.

The approximation looks suspiciously close to hitting some low mines, but a quick test shows it does not. Only 38 mines need to be tested every time anything is done to the approximation; this part of the process is amenable to automation in the future.

B. The hexic and decic approximations

Fig. **??** shows navigation through some difficult territory. The hexic polynomial covers over 700 million possible input values of the approximately one billion inputs in the interval $[0, 1]$.

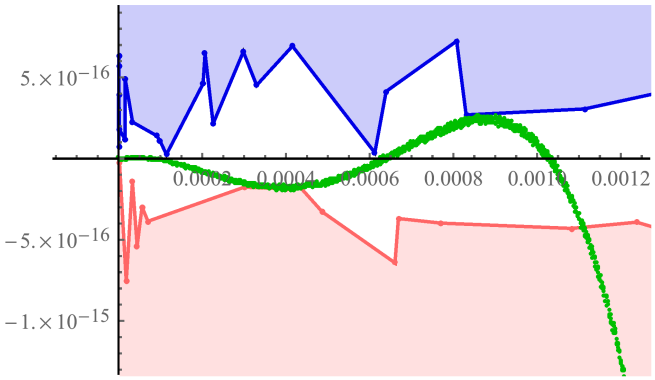

Fig. 7. Cubic approximation evaluated with fast fixed-point arithmetic

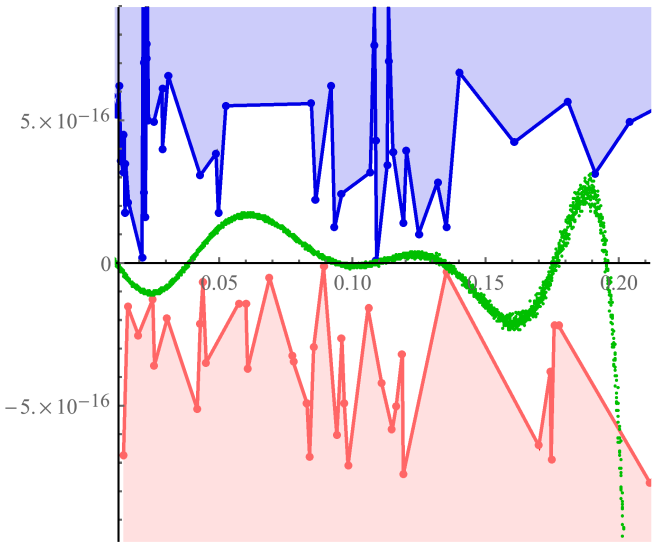

Fig. 8. Mine-avoiding fixed-point hexic polynomial

A rule of thumb seems to be to use an approximation with about as many degrees of freedom (coefficients to adjust) as there are mines to avoid in the range traversed by the approximation. The decic polynomial that finishes the coverage of the $[0, 1]$ interval is sufficiently accurate that we have little trouble steering it through the minefield; that margin allows the fixed point arithmetic to use somewhat smaller integers since there is room to allow blurring, as shown in Fig. **??**.

This covers posits in all binades from the smallest representable positive real up to 1.0. The minefield for $-1 \leqslant x \leqslant 0$ is different from the minefield from the one for $0 \leqslant x \leqslant 1$ and we must find approximations to cover that region separately; A similar strategy of cubic, hexic, and decic polynomials suffices.

C. Assembling the pieces

For floats, the density of values is repeated in every binade except for the subnormal values. For posits, the density of representable reals greater than 1 is halved every time the

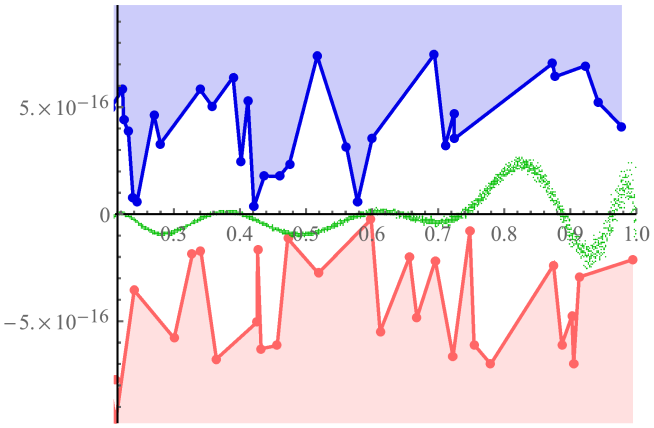

Fig. 9. Mine-avoiding fixed-point decic polynomial

domain increases by a factor of *useed*; for 32-bit values, $useed = 2^{2^2} = 16$ in the Draft Standard [**?**]. The minefields for binades in $[1, maxreal]$ and $[-maxreal, -1]$ are perfect subsets of the ones that cover $[-1, 0]$ and $[0, 1]$ so there is no need to watch out for additional mines to cover the number ranges from 1 to *maxreal* and $-maxreal$ to -1. Fig. **??** shows the composite approximating function; a binary decision tree of minimal depth is used to determine which region an input value lies in.

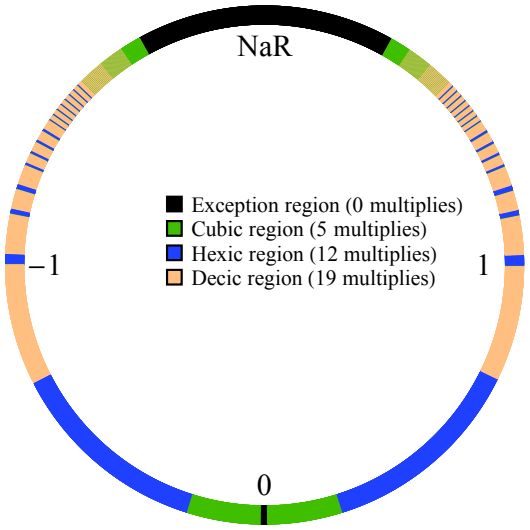

Fig. 10. Composite perfectly-rounded function for complete 32-bit posit ring

The average timing depends on the distribution of input values, of course, but the variation is small. If all inputs were equally likely, the average cost would be that of the hexic case. At the time of this writing, the routine has only been proved in *Mathematica* and is still being ported to C; however, our estimate is that it will execute in 50–70 clock cycles, uniformly. While other correctly-rounded libraries for float arithmetic have focused on 64-bit precision [**?**], [**?**], [**?**], we suspect the minefield approach will prove to be quite competitive once direct comparison is possible for the same formats and precisions.

V. The Power Function y^w

While the methods of V. Lefèvre et al. seem up to the task of finding all the hard-to-round points for transcendental functions of a single variable up to 64-bit precision [**?**], the problem of two-input transcendental functions seems to make their challenge hopeless since the search space is many orders of magnitude larger. This has led Kahan to make the following oft-repeated quote about seeking exact rounding for the power function y^w:

> "Nobody knows how much it would cost to compute y^w correctly rounded for every two floating-point arguments at which it does not over/underflow. Instead, reputable math libraries compute elementary transcendental functions mostly within slightly more than half an ulp and almost always well within one ulp. Why can't y^w be rounded within half an ulp like SQRT? Because nobody knows how much computation it would cost... No general way exists to predict how many extra digits will have to be carried to compute a transcendental expression and round it correctly to some preassigned number of digits. Even the fact (if true) that a finite number of extra digits will ultimately suffice may be a deep theorem [**?**]."

Consider the following theorem:

Theorem 1 (y^w is algebraic for floating-point y and w): Let y and w be numbers of the form $y = j \times b^m$ and $w = k \times b^n$ where b is an integer greater than 1, and j, k, m, and n are integers. Then y^w is algebraic, not transcendental.

Proof: We can always express y and w as $y = \frac{j}{b^M}$ and $w = \frac{k}{b^N}$ where M and N are nonnegative integers. Then $x = y^w = (j/b^M)^{k/b^N}$ means that $x^{b^N} = (j/b^M)^k = j^k/b^{kM}$. If $k \geqslant 0$, then $b^{kN}x^{b^N} - j^k = 0$; if $k < 0$, then $j^{-k}x^{b^N} - b^{-km} = 0$. In both cases, x is the root of a polynomial with integer coefficients, which by definition means x is algebraic, not transcendental. ■

Since fixed-point and posit formats are also of the form of an integer times a base raised to an integer power, the theorem applies to them as well. The function e^x is transcendental for rational x since the base e is transcendental. Hilbert's 7th Conjecture (resolved by Gelfond) says y^w is transcendental when $y \neq 0, 1$ is algebraic and w is irrational [**?**]. Kahan's mistake may stem from an unstated assumption that the only way to compute y^w is by using log and exp functions, when y^w is actually the root of a (possibly very large) integer.

The proof can be used in conjunction with the Minefield Method to determine whether a calculation of y^w is correctly rounded, and do so with resources and work that are bounded and known in advance. It is still a difficult function to get right, but it is *not* unreasonable to ask that a computing environment guarantee correct rounding for the power function, for every possible input. The following example shows why this is; again

we use IEEE binary16 floats to make the numbers easier to read, but the reasoning applies for any precision.

Suppose $y = \frac{1025}{1024} = 1.0009765625$ and $w = \frac{1535}{1024} = 1.4990234375$, both exactly expressible as binary16 floats. The value y^w is close to the high mine tie point, which is halfway between the two floats $\frac{1025}{1024}$ and $\frac{1026}{1024}$ separated by one ULP. In other words, this instance of y^w is hard to round correctly. Suppose an approximation tells us that y^w should round to $\frac{1025}{1024}$ and we wish to check this. The hypothesis is

$$\frac{1025 - 1/2}{1024} < (1025/1024)^{1535/1024} < \frac{1025 + 1/2}{1024}$$

which simplifies to the purely integer computation

$$2049^{1024}2048^{511} < 2050^{1535} < 2051^{1024}2048^{511},$$

and an extended-precision integer calculation shows that both inequalities are true. The number of bits is bounded and known in advance; here, all three integers are about two kilobytes in binary, making the test expensive but not intractable. In practice, only the first few bits of the integer powers would be needed to make the decision. Algorithm refinement and experimentation is needed to determine if this approach can lead to a method for correctly-rounded y^w that is of uniform high speed.

VI. Summary

The minefield approach described here creates a fast method based on a high-accuracy method that first finds the hard-to-round points. The points can also be found from tables in the literature. Once those points are known, the minefield of tie points minus the exact function is plotted at a scale that makes visible how to create approximations that avoid rounding in the wrong direction. Much of the approximation design process is manual at this point, not automatic. Through the use of 64-bit integer operations instead of floating-point instructions, the resulting routines are quite fast (a few tens of clock cycles on current microprocessors), and more importantly, the speed is uniform instead of requiring special efforts for hard-to-round points.

The long-standing tradeoff between (uniform) high speed and everywhere-correct rounding for transcendental functions of one variable may be at an end; it is possible to have both, at the cost of considerable human effort in the creation of the approximating functions. The power function, long thought to be a case of a hard-to-round transcendental, is actually an algebraic number involving large integers and roots, which seems to point a way to make its correct rounding provable at practical speed.

References

[1] Lefèvre, V., Muller, J.-M.: Worst cases for correct rounding of the elementary functions in double precision. Proc. of the 15th IEEE Symposium on Computer Arithmetic (Jun. 2001).

[2] 754-2019: IEEE Standard for Floating-Point Arithmetic. Online. Available: https://standards.ieee.org/standard/754-2019.html.

[3] Ziv, A. et al.: LIBULTIM 3. Online. Available: http://www.math.utah.edu/cgi-bin/man2html.cgi?/usr/local/man/man3/libultim.3 (2008)

[4] Sun Microsystems. Online. Available: LIBMCR 3. http://www.math.utah.edu/cgi-bin/man2html.cgi?/usr/local/man/man3/libmcr.3. (2008)

[5] Daramy-Loirat, C. et al.: CR-LIBM: A library of correctly rounded elementary functions in double-precision Online. Available: https://hal-ens-lyon.archives-ouvertes.fr/ensl-01529804/file/crlibm.pdf

[6] Cody, W.J., Waite, W.: Software Manual for the Elementary Functions. Prentice-Hall, Englewood Cliffs NJ (1980).

[7] Beebe, N.H.F.: The Mathematical-Function Computation Handbook. Springer, Cham Switzerland (2018).

[8] Posit Working Group: Posit Standard Documentation Release 3.2-draft June 23, 2018. Online. Available: http://www.math.utah.edu/cgi-bin/man2html.cgi?/usr/local/man/man3/libultim.3.

[9] Kahan, W.: A logarithm too clever by half. Online. Available: https://people.eecs.berkeley.edu/ wkahan/LOG10HAF.TXT.

[10] Kahan, W., Darcy, J.D.: How Java's Floating-Point Hurts Everyone Everywhere. Online. Available: https://people.eecs.berkeley.edu /∼wkahan/JAVAhurt.pdf.

[11] Andrysco, M. et al.: On Subnormal Floating Point and Abnormal Timing. Proc. of the 2015 IEEE Symposium on Security and Privacy, pp. 623–639 (May 2015).

[12] Muller, J.-M. et al.: Handbook for Floating-Point Arithmetic, Section 10.5, Second Edition. Springer, Cham Switzerland (2018).

[13] Gelfond, A.O.: Transcendental and algebraic numbers. Dover Phoenix editions, New York (2002) [1952].