Decimal Transcendentals via Binary

John Harrison
Intel Corporation, JF1-13
2111 NE 25th Avenue
Hillsboro OR 97124, USA
Email: johnh@ichips.intel.com

Abstract

We describe the design and implementation of a comprehensive library of transcendental functions for the new IEEE decimal floating-point formats. In principle, such functions are very much analogous to their binary counterparts, though with a few additional subtleties connected with 'scale' (preferred exponent). But our approach has been not to employ direct techniques, but rather to re-use *existing binary functions as much as possible, both for greater efficiency and ease of implementation. For some functions the most straightforward approach (convert from decimal to binary, perform binary operation, convert back) works well. In many cases, however, these are insufficiently accurate, and subtler approaches must be used.*

1. Introduction

The latest revision of the IEEE 754 Standard for binary floating-point arithmetic [5] includes new decimal formats. Intel is supporting these formats using a comprehensive open-source software library, written in portable ANSI C and compatible with numerous architectures (not just Intel's own), compilers and operating systems. It is available via the URL 'http://software.intel.com/en-us/articles/intel-decimal-floating-point-math-library'.

A significant requirement for this library is a set of decimal transcendental functions ($\sin$, $\log$ etc.). Except for *scale*, which leads to multiple representations of most decimal numbers (1.0 is distinct from 1.00 etc.), the specifications of binary and decimal floating-point arithmetic are analogous. So it would seem quite natural to take the well-established techniques for binary floating-point special functions [6] and transfer them, mutatis mutandis, to implement the decimal version. However, there are two significant disadvantages to this, one technical and one business-related:

- For most target platforms, basic decimal operations like addition and multiplication that would be used in such an implementation are implemented in software and are substantially slower than their binary counterparts, which are implemented in hardware. Even current decimal hardware implementations we know about do not yet match the speed of their binary counterparts [3]. Of course, there are platforms without even binary floating-point hardware, but these are increasingly unusual.
- Implementing the library from scratch would be a major undertaking and difficult to do quickly, since all aspects (tables, polynomial coefficients, precise accuracy needs) are different in detail from the binary designs. It's also not clear whether the importance yet justifies the allocation of resources, since few transcendental functions are used in most of the financial applications at which decimal is aimed.

Both of these considerations motivate an alternative approach: simply implementing each decimal function in terms of a corresponding binary one. We already have such binary functions, often highly optimized, in our existing mathematical libraries. We also already have conversions, and correctly rounded ones at that, between the decimal and binary floating-point formats [2]. So at the simplest we could consider an almost trivial implementation:

1) Convert the input(s) from decimal to binary
2) Compute the binary transcendental function
3) Convert the result from binary to decimal

Since there are several decimal formats to be supported, and several binary ones that we already have functions for, we need to choose which binary format to use for a particular decimal format. There are two main considerations in the choice, precision and range. The precision issues are the main subtlety to which this paper is devoted, and we will discuss this in much greater detail later. In short, our choices are motivated by the natural intuition that the binary format should be at least as accurate as the decimal format if we want to achieve reasonable accuracy, say of the order of an ulp (unit in the last place). A similar desideratum applies to range: the simple approach may break down if the range of numbers in the decimal format exceeds the range in the corresponding binary format. There are two distinct problems, caused by a finite nonzero decimal number either overflowing to infinity or underflowing to zero when converted to binary. In some cases we are lucky, and things will work out anyway even in the presence of overflow and/or underflow, because of the standard behavior of the binary functions on zeros or infinities. In general, however, it is strongly desirable for the range of the binary format to

Formats	Precision (bits)	Ranges
`decimal32` / `binary64`	23.25 / 53	$10^{-101} \ldots 10^{97}$ / $10^{-324} \ldots 10^{308}$
`decimal64` / `binary80`	53.15 / 64	$10^{-398} \ldots 10^{385}$ / $10^{-4951} \ldots 10^{4932}$
`decimal128` / `binary128`	112.95 / 113	$10^{-6176} \ldots 10^{6145}$ / $10^{-4966} \ldots 10^{4932}$

Table 1. Binary formats used to implement a given decimal format

at least match the range of the decimal format.

In brief, we use the correspondences indicated in Table 1. It can be seen that we meet our desiderata comfortably for both the `decimal32` and `decimal64` formats, where the corresponding binary formats offer substantially higher range and precision. In the case of `decimal128`, we use `binary128` (quad), the highest binary precision available without using general-precision libraries like MPFR (http://www.mpfr.org/). Yet the precision of the binary format is only marginally above that of the decimal format, and the range of numbers that it can represent is actually markedly smaller. This means that we will usually have to work significantly harder to realize our `decimal128` functions than the others.

2. Error propagation

Consider implementing a unary transcendental function f by converting to binary, performing a binary transcendental operation and then converting back. Suppose that the initial decimal-to-binary conversion gives a relative error of δ, that the binary transcendental gives a relative error of ϵ and that the final binary-to-decimal conversion gives a relative error of η. The overall result is then

$$f(x[1+\delta])(1+\epsilon)(1+\eta)$$

Some error with the same typical order as η is inevitable when producing a result in our chosen decimal format, so we are not concerned about that. As for ϵ, for good-quality binary transcendentals this is typically going to be only a little over 0.5 ulp for the intermediate binary format. Since this is always more accurate than the decimal format, and usually much more accurate, we also consider this error acceptable. Our only worry is the way in which the initial error in decimal-to-binary conversion propagates through the function f. The relative error this causes in the result is about:

$$f(x[1+\delta])/f(x)-1 \approx (f(x)+f'(x)\delta x)/f(x)-1 = \frac{xf'(x)}{f(x)}\delta$$

In other words, the relative error δ in the conversion into the binary format, at most 0.5 ulps in that format, gets amplified by the condition number $\left|\frac{xf'(x)}{f(x)}\right|$ as usual in numerical methods [4].

The acceptability of the naive algorithm now depends on whether this amplification is enough to overwhelm the generally higher accuracy of the intermediate binary format. We will later consider this case-by-case for various important functions and hence determine when the naive algorithm does or does not work well. But there are certainly situations where it does not, so let us consider how to refine the approach when this happens.

We have been thinking of the initial decimal-to-binary conversion as a black box. But it is actually rather easy to write a variant that, given a decimal input x, returns a two-part binary result (x_{hi}, x_{lo}) such that $x \approx x_{hi} + x_{lo}$ to about *twice* the binary working precision. Our existing implementation [2], in order to ensure correct rounding according to the IEEE specification, already creates an intermediate result of this sort of accuracy, and intercepting the low part of this and returning it as a binary number adds almost nothing to the runtime. (This also suggests that in the cases where the naive approach *does* work well, we could use a 'quick and dirty' version of the decimal-to-binary conversion that is not correctly rounded but is only slightly worse than that, and still monotonic. There would be little net difference in the quality of the overall results, but it could potentially be significantly faster. This is on our list for future optimizations.)

Now that we have a much more accurate binary representation of the decimal input, we can try to compensate for the error x_{lo} in decimal-to-binary conversion. We have

$$f(x) \approx f(x_{hi} + x_{lo}) \approx f(x_{hi}) + f'(x_{hi})x_{lo}$$

and so if we can compute the derivative $f'(x_{hi})$ effectively, we can obtain a large improvement in the intermediate accuracy, though we sometimes need a careful organization to incorporate it into the decimal result.

In a few rare cases we have no simple expression for the derivative (see the discussion of the Γ-function `tgamma` below). In such cases we can sometimes evaluate the binary function at two adjacent floating-point values that straddle the decimal input and then use x_{lo} to interpolate between them, e.g. if $x_{hi} \le x \le x_{hi}^+$ using

$$f(x) \approx f(x_{hi}) + \frac{x_{lo}}{(x_{hi}^+ - x_{hi})}(f(x_{hi}^+) - f(x_{hi}))$$

We could even consider still more general methods with nonlinear interpolation from several values of the binary function, but this has never been useful in our applications. In all but the most difficult cases, the basic collection of tricks turns out to be enough for our purposes. We will note below the relatively few cases where even all this seems

inadequate and we needed, at least in part, to produce a custom decimal implementation.

3. Easy cases

We will first consider some easy cases where the 'naive' algorithm, perhaps supplemented by some simple special-case code, suffices.

Arctangent

Here the condition number is

$$\frac{xf'(x)}{f(x)} = \frac{x}{(1+x^2)\,\mathrm{atan}(x)}$$

This is perfectly well-behaved, peaking at 1 around $x = 0$ and elsewhere being < 1 in magnitude, as shown in Figure 1. This means that a "naive" algorithm is almost entirely satisfactory. For `decimal128` we still need to confront the fact that the range of quad is more limited than that of `decimal128`, and take some special measures:

- For very small arguments (we use this for $|x| < 10^{-40}$) we have $\mathrm{atan}(x) \approx x - x^3/3$. We could simply return x, but for a touch more refinement in directed rounding modes we actually perform a decimal `fma` calculation $x - 10^{-40} \cdot x$ which will round exactly as $x - x^3/3$ and is somewhat more efficient. Of course, our function is by no means correctly rounded in the main part of the range, so it is not unreasonable to neglect this issue here, and we sometimes do for other functions.
- For very large arguments, no special measures are needed. We just rely on the main path, which when the decimal-to-binary conversion overflows will return $\mathrm{atan}(\pm\infty) = \pm\pi/2$, a value that will then convert back correctly. This will likewise work in the best way in directed rounding modes *provided* that the underlying binary function is rounded to nearest, as we presently assume, for this binary rounding and the exact value are not separated by a decimal rounding boundary. We anticipate in the future having common rounding modes for binary and decimal, in which case a little more refinement would be needed in this calculation. Besides, it may be more efficient to intercept and deal specially with very large inputs instead of going through the main path.

Error function

The error function is defined as

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

so its derivative is

$$\mathrm{erf}'(x) = \frac{2}{\sqrt{\pi}} e^{-x^2}$$

and therefore the condition number is

$$\frac{x\,\mathrm{erf}'(x)}{\mathrm{erf}(x)} = \frac{2xe^{-x^2}}{\sqrt{\pi}\mathrm{erf}(x)}$$

This is a well-behaved function, peaking at 0 with absolute magnitude ≤ 1, so we can afford the naive algorithm, modulo special cases. The binary function has $\mathrm{erf}(\pm\infty) = \pm 1$, so as with the arctangent, inputs that overflow in quad work by default. This time, however, we need more care over *underflow* in the `decimal128` case when the input is converted to quad. For x very small in magnitude, we have

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \approx \frac{2}{\sqrt{\pi}} \int_0^x 1 dt = \frac{2}{\sqrt{\pi}} x$$

so we just perform a decimal multiplication of the input by a pre-stored decimal rounding of $\frac{2}{\sqrt{\pi}}$

Hyperbolic tangent

Here again we have a well-behaved function, with

$$\frac{xf'(x)}{f(x)} = \frac{x\,\mathrm{sech}(x)^2}{\tanh(x)}$$

having a peak of 1 at $x = 0$ and being < 1 in magnitude elsewhere. In fact, this function is remarkably close to erf in its general behavior, and their graphs could easily be mistaken at a quick glance. For very small x, however, the limiting behavior is just $\tanh(x) \approx x - x^3/3$, so we deal with it as with the arctangent.

4. Moderate cases

We now consider some cases where more non-trivial processing of the input is needed, at least in some parts of the domain.

Inverse sine and cosine

For inverse sine, the condition number is

$$\frac{xf'(x)}{f(x)} = \frac{x}{\sqrt{1-x^2}\,\mathrm{asin}(x)}$$

and for inverse cosine:

$$\frac{xf'(x)}{f(x)} = \frac{x}{\sqrt{1-x^2}\,\mathrm{acos}(x)}$$

Both these values are moderate for most $|x| \ll 1$, and in particular are ≤ 1.5 in magnitude for $|x| \leq 0.7$. However, in each case there is a singularity at $x = \pm 1$ around which the condition is unbounded, and this is particularly steep in the case of $\mathrm{acos}(x)$ for $x = 1$; see Figure 2. Thus, at least in the case of the `decimal128` format, we need to take special measures for inputs close to ± 1.

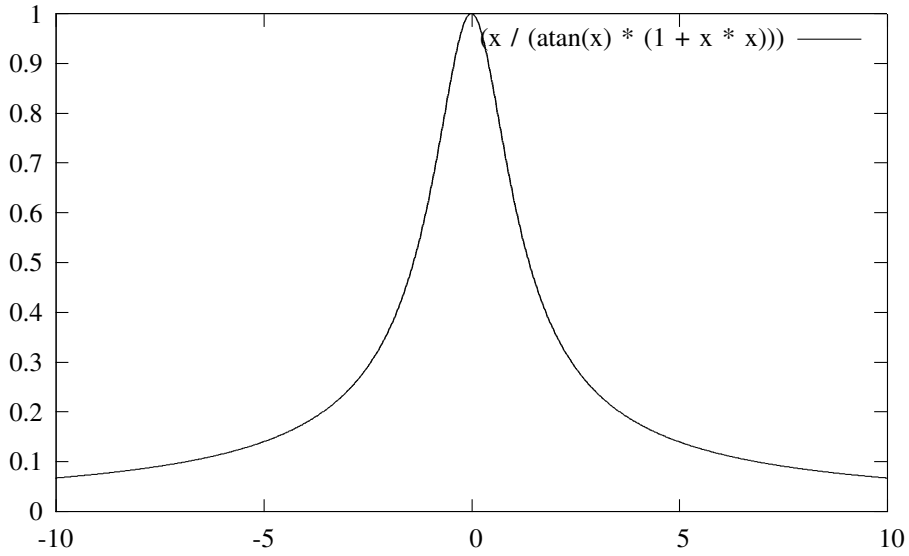

Figure 1. Condition of $\mathrm{atan}(x)$

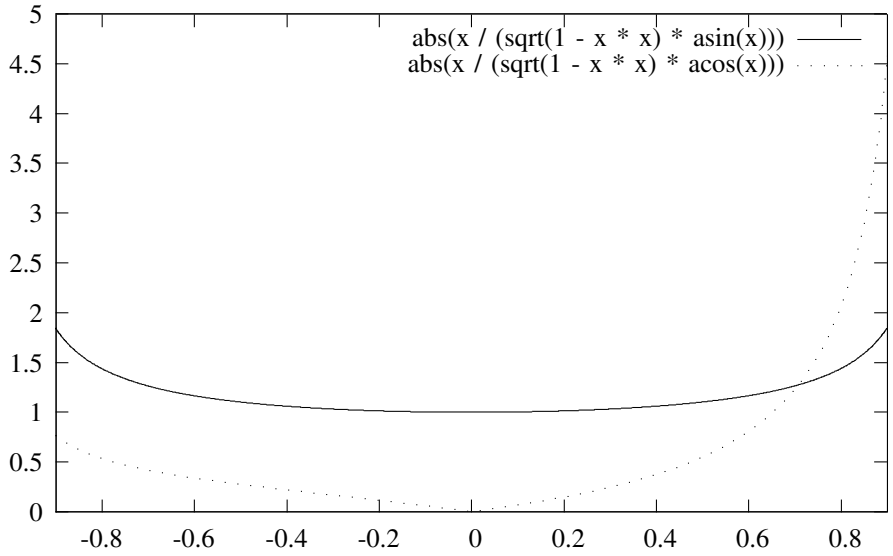

Figure 2. Condition of $\mathrm{asin}(x)$ and $\mathrm{acos}(x)$

For such inputs we exploit some relationships between the functions. The simplest such relationship is $\mathrm{asin}(x) + \mathrm{acos}(x) = \pi/2$, but this is not all that useful for the control of numerical errors. Instead we use the relationships

$$\mathrm{asin}(x) = \begin{cases} \mathrm{acos}\sqrt{1-x^2} & \text{if } x \geq 0 \\ -\mathrm{acos}\sqrt{1-x^2} & \text{if } x \leq 0 \end{cases}$$

and

$$\mathrm{acos}(x) = \begin{cases} \mathrm{asin}\sqrt{1-x^2} & \text{if } x \geq 0 \\ \pi - \mathrm{asin}\sqrt{1-x^2} & \text{if } x \leq 0 \end{cases}$$

We can compute $1 - x^2$ with good relative error using a single decimal `fma`, and the rest of the computation can then be done in a straightforward way since we are back in the well-conditioned region of the function on the right-hand side.

Logarithm

The condition number here is $1/\log(x)$:

$$\frac{xf'(x)}{f(x)} = \frac{xx^{-1}}{\log(x)} = 1/\log(x)$$

This has a singularity at $x = 1$ near which it is unbounded. Admittedly, floating-point numbers cannot come all that close to 1, but with d decimal digits the closest approach is $1 - 10^{-d}$ in which case the condition number is about 10^d.

- Implementing `decimal32` via double precision, the condition number is $< 10^7$. Since $|\delta| \leq 2^{-53}$, the net relative error is $\leq 10^7 \cdot 2^{-53} \cdot 10^7 \approx 0.011$ ulps, well within acceptable limits.
- Implementing `decimal64` via double-extended precision, the condition number is $< 10^{16}$. Since $|\delta| \leq 2^{-64}$, the net relative error is $\leq 10^{16} \cdot 2^{-64} \cdot 10^{16} \approx 10^{13}$ ulps, almost certainly unacceptable.
- Implementing `decimal128` via quad precision, the condition number is $< 10^{34}$. Since $|\delta| \leq 2^{-113}$, the net relative error is $\leq 10^{34} \cdot 2^{-113} \cdot 10^{34} \approx 10^{34}$ ulps, clearly unacceptable.

Thus, while we can implement the `decimal32` version 'naively', we need special measures near $x = 1$ for the wider formats. The special measures consist in tracking more carefully the error when converting the decimal input x to its binary approximation x^*. We do this for inputs where $|x - 1| < 0.5$ by computing both $y = x - 1$ as a decimal operation and $y^* = x^* - 1$ as a binary operation. By the usual cancellation properties, these are both exact, so we can convert y to binary and subtract y^* to get a binary correction term e so that $x \approx x^* + e$ to extra precision. We then correct the naive computation by e/x^*, exploiting

$$\log(x^* + e) = \log(x^*(1 + e/x^*)) \approx \log(x^*) + e/x^*$$

In the case of `decimal128`, we also have to worry about overflow or underflow when converting to the input to quad. Those, however, we deal with straightforwardly by a decimal multiplication of the input by $10^{\pm 4464}$ and a subsequent correction of the result by $\mp 4464 \log 10$. The value 4464 was chosen because the error from storing $4464 \log 10$ in quad is particularly small.

An alternative approach to the problems near $x = 1$, pointed out by one of the reviewers, would be to compute $y = x - 1$ in decimal as now but then convert y to a binary counterpart y^* and use the `log1p` function to compute $\log(1 + y^*)$. This is certainly somewhat simpler and may well be more efficient.

5. Relatively difficult cases

We now consider cases where the naive algorithm needs quite elaborate precautions, and we therefore find it easier to use the 2-part conversion from decimal to binary.

Exponential function

The condition number here is x:

$$\frac{xf'(x)}{f(x)} = \frac{xe^x}{e^x} = x$$

Thus, the relative error swells by a factor of x, which may be quite large. On the other hand, the range of x is quite limited because the function overflows fairly quickly, so the results are not completely catastrophic:

- Implementing `decimal32` via double precision, the maximal x before overflow in `decimal32` is $\log(10^{97}) < 224$. Since $|\delta| \leq 2^{-53}$, the net relative error is $\leq 224 \cdot 2^{-53} \cdot 10^7 \approx 2.49 \times 10^{-7}$ ulps, almost negligible.
- Implementing `decimal64` via double-extended precision, the maximal x before overflow in `decimal64` is $\log(10^{385}) < 887$. Since $|\delta| \leq 2^{-64}$, the net relative error is $\leq 887 \cdot 2^{-64} \cdot 10^{16} \approx 0.481$ ulps, almost certainly within acceptable limits.
- Implementing `decimal128` via quad precision, the maximal x before overflow in `decimal128` is $\log(10^{6145}) < 14150$. Since $|\delta| \leq 2^{-113}$, the net relative error is $\leq 14150 \cdot 2^{-113} \cdot 10^{34} \approx 13626$ ulps. We do not consider this acceptable.

Special measures are certainly needed in the case of `decimal128`. Using only a 'black box' decimal-to-binary conversion, the best approach we can think of involves separating the input into two pieces $x = h + l$ where h can be converted exactly to binary. (One approach is simply to let h be the rounding of x to the nearest integer, but it's more accurate to let h be of the form $N/2^k$ for moderate k that ensures exact representability in both binary and decimal formats.) However, using the accurate 2-part conversion from decimal to binary, things are reasonably straightforward since we already get a decomposition $x = h + l$ where both h and l are binary numbers with $l \ll h$, and we can compute

$$e^x = e^{h+l} = e^h e^l \approx e^h(1 + l)$$

The only additional complication for `decimal128` comes in the fact that for inputs that are reasonably large in magnitude, the quad exponential function may suffer overflow or underflow in its results. We avoid this by checking the size of h first, and either adding or subtracting 11000 if necessary, postcorrecting the result in a final decimal multiplication by e^{11000}.

Complementary error function

The complementary error function is defined as

$$\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$$

so its derivative is

$$\mathrm{erfc}'(x) = -\mathrm{erf}'(x) = \frac{-2}{\sqrt{\pi}} e^{-x^2}$$

and therefore the condition number is

$$\frac{x\,\mathrm{erfc}'(x)}{\mathrm{erfc}(x)} = \frac{2xe^{-x^2}}{\sqrt{\pi}\mathrm{erfc}(x)}$$

Despite the fact that $\mathrm{erf}(x)$ and $\mathrm{erfc}(x)$ have such a close relationship, the difference that $\mathrm{erfc}(x) \to 0$ as $x \to \infty$ means that the condition becomes very large as x becomes large and positive (Figure 3). We note the asymptotic formula [1]:

$$\mathrm{erfc}(x) \approx \frac{e^{-x^2}}{\sqrt{\pi}}(\frac{1}{x} - \frac{1}{2x^3} + \frac{3}{4x^5} - \frac{15}{8x^7} + \cdots)$$

This implies that for large x, the condition is about $2x^2$, which is quite bad. On the other hand, x doesn't have to be all that large before e^{-x^2} is so tiny that it will underflow to zero, so the whole range needn't be tackled:

- For `decimal32`, we underflow for $\frac{e^{-x^2}}{\sqrt{\pi}x} < 10^{-101}$, i.e. for about $x \geq 16$, at which point the condition is about 832, nowhere near enough to erase the `binary64-decimal32` advantage.
- For `decimal64`, we underflow for $\frac{e^{-x^2}}{\sqrt{\pi}x} < 10^{-398}$, i.e. for about $x \geq 31$, at which point the condition is about 1922. Since $1922 \cdot 2^{-64} \approx 10^{-16}$, this is probably just about acceptable.
- For `decimal128`, we underflow for $\frac{e^{-x^2}}{\sqrt{\pi}x} < 10^{-6176}$, i.e. for about $x \geq 120$, at which point the condition is about 28322. This would not be acceptable with the naive algorithm.

However, the condition for `decimal128` is moderate enough that we can use the 2-part conversion $x = h + l$ and make a postcorrection where necessary using the derivative $\mathrm{erfc}'(x) = \frac{-2}{\sqrt{\pi}}e^{-x^2}$ via $\mathrm{erfc}(x) = \mathrm{erfc}(h) - \frac{2}{\sqrt{\pi}}e^{-x^2}l$.

We actually split the domain into four different cases. First of all, if $x \leq 0$ the naive computation is well-conditioned and we just proceed that way. Otherwise, if $0 \leq x \leq 105$, we just correct the naive result using the derivative computed in binary by $\mathrm{erfc}(x) = \mathrm{erfc}(h) - \frac{2}{\sqrt{\pi}}e^{-h^2}l$. If $x \geq 120$, the result always underflows to zero, though as usual we do a dummy calculation for the sake of the directed rounding modes. The most difficult case is for $105 \leq x \leq 120$, where the binary exponential would underflow; in this case we essentially use the asymptotic expansion noted above. (We truncate at the twelfth term.) To add to the difficulty, even the computation of e^{-x^2} itself is illconditioned, and we need to keep extra precision when squaring, again using the decimal `fma` operation to get a 2-part product. However, the main terms of the asymptotic expansion can be done fairly directly using quad.

6. Difficult cases

We now come to the cases where all our tricks seem inadequate and we need, at least to some degree, a custom implementation.

Trigonometric functions

The trigonometric functions are all highly ill-conditioned near various large multiples of $\pi/2$, so we did not even explore the possibility of using the full binary function as a subroutine. Instead we have a 2-part structure:

- A custom decimal implementation of trigonometric range reduction to give $x = N\pi/2 + r$ with $|r| \leq \pi/4 + \epsilon$.
- A naive use of the underlying binary algorithm to compute the trigonometric function of the reduced argument r.

The argument reduction is a variant of the traditional approach [7], implemented in decimal. The latter part, and the subsequent reconstruction of the final result, presents no great difficulties. For example

$$\sin(x) = \begin{cases} \sin(r) & \text{if } N \equiv 0 \pmod 4 \\ \cos(r) & \text{if } N \equiv 1 \pmod 4 \\ -\sin(r) & \text{if } N \equiv 2 \pmod 4 \\ -\cos(r) & \text{if } N \equiv 3 \pmod 4 \end{cases}$$

Since the underlying trigonometric functions $\sin(r)$, $\cos(r)$ and $\tan(r)$ are well-conditioned for $|r| \leq \pi/4+\epsilon$, everything beyond the range reduction can be done 'naively'. The reduced argument r is directly extracted from the large-integer arithmetic in the decimal range reduction as a single `binary128` number and the usual binary function and binary-to-decimal conversion are applied.

The power function

Since the power function has two arguments, both of which may be perturbed when converting decimal arguments to binary, we analyze the condition separately for each argument.

- For x we have:

$$\begin{aligned} d(x^y)/dx &= d(e^{y\log(x)})/dx \\ &= e^{y\log(x)}d(y\log(x))/dx \\ &= x^y y/x \end{aligned}$$

  so the condition number is:

$$x(x^y y/x)/x^y = y$$

- For y we have:

$$\begin{aligned} d(x^y)/dy &= d(e^{y\log(x)})/dy \\ &= e^{y\log(x)}d(y\log(x))/dy \\ &= x^y \log(x) \end{aligned}$$

  so the condition number is:

$$y(x^y \log(x))/x^y = y\log(x)$$

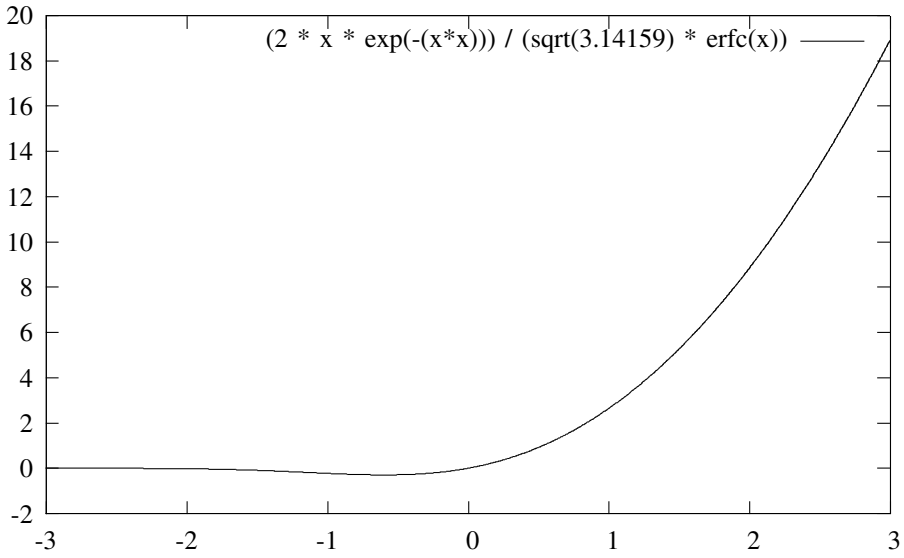

Figure 3. Condition of erfc(x)

As expected, in the case $x = e$ we get y, as with the exponential function.

If we use Δ to denote relative change, we therefore have roughly

$$\Delta(x^y) = y\Delta(x) + y\log(x)\Delta(y)$$

Thus, things are worse when y is large, as with the exponential function. The range of y is limited by the overflow threshold B: we must have $x^y < B$, i.e. $y < \log(B)/\log(x)$. This limit is weakest when $x \approx 1$ since then $\log(x)$ is close to zero; in this case the relative error is dominated by $y\Delta(x) = (\log(B)/\log(x))\Delta(x)$. Thus, the effects are similarly bad to the case of the logarithm, but with the additional scaling by $\log(B)$. This is even enough to cause appreciable inaccuracy in the case of `decimal32`:

- Consider $(1 + 10^{-6})^{237166383}$, which is just below the overflow threshold. The actual relative error in rounding $1 + 10^{-6}$ to double precision is about 8.227×10^{-17}. The relative error in the result is about $(1 + 8.227 \times 10^{-17})^{237166383} - 1 \approx 2 \times 10^{-8}$. This is of the order of 0.2 ulps.

Despite extensive investigation, we have not been able to arrive at a truly satisfactory way of exploiting existing binary functions. Instead, we have produced a custom implementation that roughly uses $x^y = \pm e^{y \log |x|}$. The determination of the sign takes some care, looking at the sign of x and whether y is an odd or even integer. There are also a large number of special cases to deal with, some of them a little surprising; for example the standard behavior is that $1^y = 1$ even if y is a NaN. Underlying the computation of $e^{y \log |x|}$ is our own custom implementation of a decimal logarithm function in extra-high precision. This returns a 2-part result $l_{hi} + l_{lo}$ and we likewise compute $y(l_{hi} + l_{lo})$ accurately in two pieces using fairly standard `fma` tricks. The net effect is that the power function, while accurate, is markedly slower than most of the other functions in our library.

7. Unsolved problems

The only functions that we have *not* got completely accurate implementations for are the two Γ-family functions `tgamma` and `lgamma`. In the case of `decimal128`, we simply cannot rely on the quad implementations that we have otherwise used, because they are themselves not accurate. Otherwise, `tgamma` presents only moderate difficulties in principle. The `lgamma` function, however, seems essentially to require some custom implementation work near its irregular zeros.

Gamma function

We can reduce things to positive arguments using the reflection formula:

$$\Gamma(x) = \frac{\pi}{\Gamma(1-x)\sin(\pi x)}$$

This presents no numerical problems since it is multiplicative and the exact decimal argument reduction for πx is straightforward, essentially just integer rounding. For `decimal128`, the possibility of overflow in the underlying quad function is problematic. The quad function overflows around $x = 1756$, whereas the `decimal128` version only overflows around $x = 2125$. So we use the duplication

formula to avoid inputs where this problem would strike:

$$\Gamma(2x) = \frac{2^{2x-1}}{\sqrt{\pi}}\Gamma(x)\Gamma(x+1/2)$$

The condition number for Γ is similar to the exponential function, and seems to be bounded by a moderate multiple of x. So correction using the low part after a 2-part decimal-to-binary conversion is satisfactory. This time, it seems difficult to use the derivative since this function (also called the 'digamma' function) isn't a standard part of binary math libraries. Instead we simply call the binary function twice on adjacent quad values and use the correction term to interpolate between them.

Log-gamma (`lgamma`) function

At first sight this looks simpler than Γ, but in fact it's somewhat harder because we also have to worry about the various cases where $\log|\Gamma(x)| \approx 0$, i.e. $|\Gamma(x)| \approx 1$. There are some such crossover points at the negative end. So if we naively take the logs of $\Gamma(1-x)$ and $\pi/\sin(\pi x)$ when dealing with negative cases, we get cancellation to a degree we can't really afford. Even using sophisticated interpolation based on the binary functions, our analysis indicates that this could not in principle give accurate results. In fairness, it may well be the case that few binary functions are accurate in relative terms in such cases; as we have noted, our own are certainly not.

8. Conclusions

By taking advantage of the existing binary implementations we have been able, in the space of just a few months, to produce a complete set of functions offering reasonable performance, and comparable accuracy to the binary ones. In only a relatively few cases have we needed elaborate custom implementations, and our accuracy is roughly as good as the binary functions we are using. The only case where we do not meet our accuracy goals are for the two gamma family functions, and in these cases our binary functions are not accurate either. So it is fair to say that our implementation, in accuracy terms, is not much worse than our binary functions. With the exception of the power function, we also see solid performance numbers. All in all, this is a solid first version of the decimal transcendental functions, but there is plenty of scope for further improvements. It would be interesting to compare the performance and accuracy of our approach against a 'custom' approach, if and when such an implementation becomes available.

The initial justifications we claimed for our approach were twofold: efficiency and ease of implementation. We think the results of this paper generally bear this out. However, in the case of the `decimal128` functions the first justification is less obviously applicable, since the `binary128` functions are also just a software library, albeit a well-established and quite efficient one. Moreover, not all platforms support the `binary80` type, at least efficiently, so one might want to contemplate using `binary64`, sacrificing some accuracy and facing similar issues to those we dealt with for `binary128` where the underlying binary format has comparable precision to the target decimal format.

Acknowledgements

The author would like to thank Marius Cornea for originally suggesting the use of the existing binary functions to implement decimal transcendentals, as well as the anonymous reviewers for ARITH whose comments significantly improved the final version of this paper.

References

[1] M. Abramowitz and I. A. Stegun. *Handbook of Mathematical Functions With Formulas, Graphs, and Mathematical Tables*, volume 55 of *Applied Mathematics Series*. US National Bureau of Standards, 1964.

[2] M. Cornea, J. Harrison, C. Anderson, P. T. P. Tang, E. Schneider, and E. Gvozdev. A software implementation of the IEEE 754R decimal floating-point arithmetic using the binary encoding format. *IEEE Transactions on Computers*, 58:148–162, 2009.

[3] L. Eisen, J. W. Ward, H.-W. Tast, N. Mading, J. Leenstra, C. J. S. M. Mueller, J. Preiss, E. M. Schwarz, and S. R. Carlough. IBM POWER6 accelerators: VMX and DFU. *IBM Journal of Research and Development*, 51(6):1–21, 2007.

[4] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, Philadelphia, 1996.

[5] IEEE. Standard for binary floating point arithmetic. ANSI/IEEE Standard 754-2008, The Institute of Electrical and Electronic Engineers, Inc., 2008. Revised version of original 754-1985 Standard.

[6] J.-M. Muller. *Elementary functions: Algorithms and Implementation*. Birkhäuser, 2nd edition, 2006.

[7] M. Payne and R. Hanek. Radian reduction for trigonometric functions. *SIGNUM Newsletter*, 18(1):19–24, 1983.