Decimal Transcendentals via Binary

John Harrison, Intel Corporation

ARITH-19, Portland OR
June 10, 2009 (11:00–11:30)

Why decimal transcendentals?

Intel is distributing a portable open-source library supporting the decimal formats in the newly revised IEEE-754R Standard:

`http://software.intel.com/en-us/articles/intel-decimal-floating-point-math-library`

Most transcendental functions are not widely used in financial applications for which decimal arithmetic is intended.

Why decimal transcendentals?

Intel is distributing a portable open-source library supporting the decimal formats in the newly revised IEEE-754R Standard:

`http://software.intel.com/en-us/articles/intel-decimal-floating-point-math-library`

Most transcendental functions are not widely used in financial applications for which decimal arithmetic is intended. But

- Some transcendentals *are* used in financial applications, e.g. computing compound interest.
- Some envisage wider use of decimal as a universal number format for typical users.
- We should have them anyway for complete IEEE-754R support.

Why via binary?

We could implement all the decimal transcendentals from scratch by modifying existing algorithms for binary functions. But:

- The underlying ‘basic’ decimal operations we would use in such an implementation are (even in current hardware) typically much slower than their binary counterparts.
- It would represent a huge amount of work: the existing binary functions have been developed and honed over many years.

This motivates the alternative approach: *re-use* the binary functions.

Ideally, we’d like to implement all the C99 (ISO/IEC 9899) transcendentals in this way, all of which already exist for binary.

Our plan

Roughly, the plan is to convert from decimal to binary, use the corresponding binary transcendental, then convert the result back to decimal.

$$\begin{tikzcd}
B(x) \arrow[r, "\text{Binary } f_B"] & f_B(B(x)) \arrow[d, "\begin{array}{c}\text{Binary to}\\ \text{decimal}\\ (D)\end{array}"] \\
x \arrow[u, "\begin{array}{c}\text{Decimal}\\ \text{to binary}\\ (B)\end{array}"] \arrow[r, "\text{Decimal } f_D"'] & \begin{array}{l} D(f_B(B(x))) \\ \approx f_D(x) \end{array}
\end{tikzcd}$$

Decimal and binary formats

Use a wider binary format than the required decimal format:

Decimal format	Binary format	Precision increase
`decimal32`	double	53 - 23.25 = 29.75
`decimal64`	double-extended	64 - 53.15 = 10.85
`decimal128`	quad	113 - 112.95 = 0.05

Range and accuracy issues

Are there any obstacles to this “naive” approach?

- In most cases the binary range is greater, but `decimal128`/quad is an exception.

- In each case the binary format is wider, but only marginally in the case of `decimal128`/quad.

The range issue for `decimal128`/quad can be a bit tedious, but extremely large or small inputs are usually easy to handle anyway.

The question of accuracy is more subtle.

Accuracy issues

We accumulate three errors in total:

$$\begin{tikzcd}
x(1+\delta) \arrow[r, "\text{Binary } f_B"] & f(x(1+\delta))(1+\epsilon) \arrow[d, "\text{Binary to decimal } (D)"] \\
x \arrow[u, "\text{Decimal to binary } (B)"] \arrow[r, "\text{Decimal } f_D"'] & f(x(1+\delta))(1+\epsilon)(1+\eta) \approx f(x)
\end{tikzcd}$$

Here ϵ and η are of the order of an ulp in the result for the binary and decimal formats respectively, and so are acceptable.

Blowup in initial conversion error

The potential problem is the error arising from the *initial* conversion from decimal to binary.

$$f(x(1+\delta)) = f(x+x\delta) \approx f(x) + f'(x)x\delta = f(x)\left(1 + \frac{xf'(x)}{f(x)}\delta\right)$$

Potential trouble arises when the condition number is much more than 1:

$$\left|\frac{xf'(x)}{f(x)}\right| \gg 1$$

If the condition number is never much more than 1, we're OK.

What if the naive approach doesn't work?

We've been considering the initial decimal-to-binary conversion as a black box.

However, since this already computes a doubly accurate intermediate result to ensure perfect rounding, it adds very little to the runtime to create a 2-part translation $x \longrightarrow x_{\mathtt{hi}} + x_{\mathtt{lo}}$.

Can use this to correct:

- If we can calculate the derivative of the function, use $f(x) \approx f(x_{\mathtt{hi}} + x_{\mathtt{lo}}) \approx f(x_{\mathtt{hi}}) + f'(x_{\mathtt{hi}})x_{\mathtt{lo}}$
- Use it to interpolate between results on two adjacent binary numbers if we can't. (Could in principle use higher-order interpolation, but this never seems useful.)

Arctangent (1)

An easy case is the arctangent function:

$$f(x) = \text{atan}(x)$$

Here the condition number is

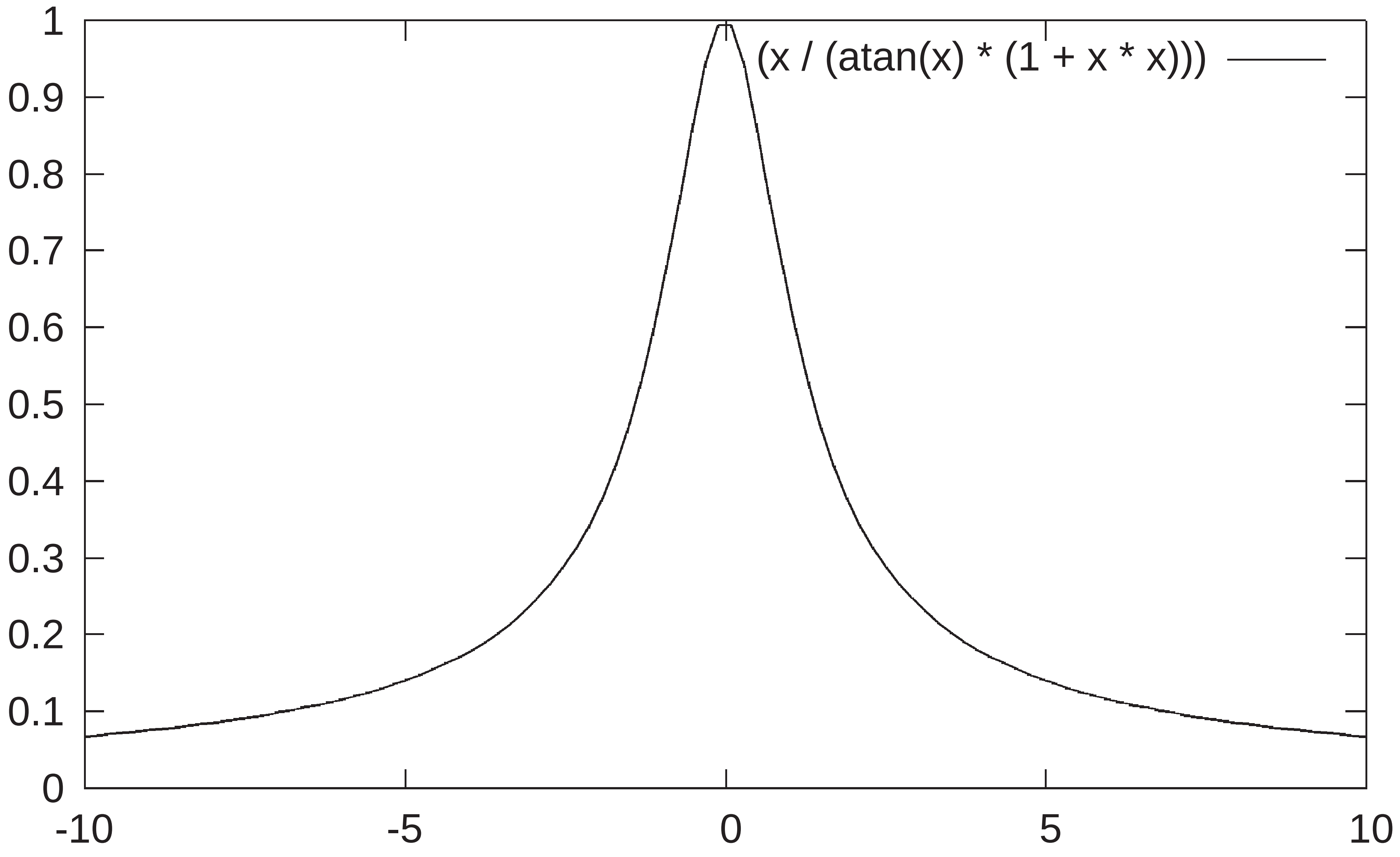
$$\frac{x f'(x)}{f(x)} = \frac{x}{(1+x^2)\text{atan}(x)}$$

This is perfectly well-behaved, peaking at 1 around $x = 0$ and elsewhere being < 1 in magnitude.

The range issue is trivial as well: we can even be lazy and propagate

$$\text{atan}(B(\pm\text{large})) = \text{atan}(\pm\infty) = \pm\pi/2$$

Arctangent (2)



Logarithm (1)

A more interesting case is the $\log$ function, where the condition is

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

This can get very large for $x \approx 1$. For a decimal format with d digits, we can get $1/\log(x) = 1/\log(1 - 10^{-d}) \approx 10^d$.

For `decimal32` and `decimal64` this is acceptable, though in the latter case it's marginal.

For `decimal128`, it's completely unacceptable, giving almost no valid bits in the worst case.

Logarithm (2)

Our implementation still uses a ‘naive’ path (after some special treatment of extreme inputs). We perform a decimal-to-binary conversion:

$$x^* = B(x)$$

and compute

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

But in the case of inputs $x \approx 1$, we perform a *decimal* subtraction

$$y = x - 1$$

and a *binary* subtraction:

$$y^* = x^* - 1$$

both of which will be exact.

Logarithm (3)

Now computing $e = B(y) - y^*$ we get an accurate low-order part, i.e.

$$x \approx x^* + e$$

Now we correct the original logarithm as follows:

$$\begin{aligned}
\log(x) &= \log(x^* + e) \\
&= \log(x^*(1 + e/x^*)) \\
&= \log(x^*) + \log(1 + e/x^*) \\
&\approx y^* + e/x^*
\end{aligned}$$

We thus obtain an accurate answer throughout the range.

Exponential

The condition number here is:

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

For `decimal32` and `decimal64` this is acceptable, but not for `decimal128`.

But we can easily use a 2-part conversion, and use a linear approximation to the derivative:

$$e^x = e^{x_{\texttt{hi}}+x_{\texttt{lo}}} = e^{x_{\texttt{hi}}} e^{x_{\texttt{lo}}} \approx e^{x_{\texttt{hi}}}(1 + x_{\texttt{lo}})$$

The power function

This is a rather nasty one since it’s ill-conditioned in various parts of its domain

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

We spent a long time trying to find ingenious ways of re-using binary, but it seems very difficult.

In the end, we based an implementation around

$$x^y = \pm e^{y\log(x)}$$

using a custom decimal logarithm function providing extra precision.

This represents the first failure of the approach of re-using binary functions.

Trigonometric functions

These are in general severely ill-conditioned close to multiples of $\pi/2$.

There seems no alternative to implementing custom range reduction in decimal.

We have implemented a slight variant of Payne-Hanek range reduction for decimal.

Decimal Payne-Hanek

For an input of $x = 10^e m$ we get

$$r[e] = (10^e/2\pi) \text{ mod } 1$$

out of a table as a binary fraction and multiply:

$$p = (m \cdot r[e]) \text{ mod } 1$$

Now shift p left two places to get the parity, and multiply the tail by $\pi/2$ to give a remainder mod $\pi/2$.

This is constructed directly as a binary floating-point number and we then apply the “naive” algorithm.

Unsolved problems

The only functions where we've failed to produce an accurate version are `tgamma` ($\Gamma(x)$) and `lgamma` ($\log|\Gamma(x)|$).

The main problem is that in neither case is our binary function accurate enough.

However, the case of `lgamma` is fundamentally harder: it has a few irregular zeros where $|\Gamma(x)| \approx 1$. Even with an accurate binary function, there's no way to get an accurate decimal one directly.

Note that this is the unique function for which the OpenCL Standard does *not* give an ulp bound!

Conclusions

- For quick implementation of a wide range of transcendentals, re-using binary seems an effective approach.
- In many cases, the “naive” approach, possibly in concert with some special tricks, gives good accuracy.
- For some difficult cases, we need to program more of the function directly in decimal.
- Nevertheless, we have a solution for all the C99 functions that is as good as binary.
- It would be interesting to optimize by using a ‘quick and dirty’ initial decimal-to-binary conversion.