Pseudo-Division Algorithms for Floating-Point Logarithms and Exponentials

Prof. W. Kahan
Univ. of Calif. at Berkeley

May 20, 2001

Abstract

Among the CORDIC-like algorithms for computing elementary transcendental functions like log and exp, certain pseudo-division algorithms are peculiarly well suited to implementation in microcode or in conjunction with software-implemented floating-point arithmetic. These algorithms need tables of comparatively modest size; they are almost as fast as the fastest digit-by-digit algorithms known; and they can achieve accuracy to within a unit or two in the last sig. bit carried. Algorithms like these are used by the Intel 8087 family of numeric coprocessors. This document is for people who wish to imitate or surpass them.

1 Introduction

In what follows, pseudo-division algorithms for $\exp(x)$, $\ln(x)$, $\exp(x)-1$ and $\ln(1+x)$ are described, and then their accuracies are explained by a crude error-analysis. Because these algorithms are intended to cohabit with binary floating-point arithmetic, the functions computed first are actually 2^x , $\log_2(x)$, $2^x - 1$ and $\log_2(1+x)$, from which the aforementioned functions are then derived. (The 2 will be dropped henceforth from $\log_2$.) The motive for this indirect approach is a belief (perhaps mistaken) that y^x might be easier to compute accurately that way. The reason for including the functions $2^x - 1$ and $\log(1+x)$ is the certainty that their high relative accuracy simplifies similarly accurate programs for certain hyperbolic and financial functions.

What makes the pseudo-division algorithms so attractive is their close resemblance to the simplest binary multiplication and division algorithms; they differ mainly in the use of varying multiplicands or divisors drawn from tables of constants. The tests, add/subtracts and shifts are otherwise very similar. But extra shifts have to be introduced to maintain full accuracy of the kind expected from floating-point operations nowadays, and these extra shifts complicate the algorithms' descriptions more than they complicate the algorithms themselves. That is why we shall begin by describing idealized algorithms that leave out the complicating extra shifts at first. Practical algorithms will be described afterwards.

2 Idealized Algorithms for $\log(x)$ and $\mathrm{txp}(x) = 2^x$

(When x is a subscripted variable, typographical limitations compel us to introduce notations like $\mathrm{txp}(x)$ for 2^x, $\mathrm{txpm1}(x)$ for $2^x - 1$, and $\mathrm{lg1p}(x)$ for $\log(1+x) = \ln(1+x)/\ln(2)$.)

What set of numbers y can satisfy an equation

$$y = 1/((1+q_1/2)(1+q_2/4)(1+q_3/8)(\ldots)(1+q_k/2^k)(1+\ldots))$$

in which every pseudo-quotient bit q_k is either 0 or 1? The smallest value y in that set must be

$$1/\eta(1) = 0.41942244\ldots \text{ where } \eta(t) = (1+t/2)(1+t/4)(1+t/8)(1+t/16)(\ldots)(1+t/2^k)(1+\ldots).$$

(An appendix, sect. n below, explains properties of $\eta(t)$.) It turns out that all values y between $1/\eta(1)$ and 1 belong to that set, many of them in more than one way. For instance

$$\begin{aligned} 2/3 &= 1/(1+1/2) \\ &= 1/((1+2^{-2})(1+2^{-3})(1+2^{-4})(1+2^{-8})(1+2^{-16})(1+...)). \end{aligned}$$

Thanks to this non-uniqueness, a relatively simple pseudo-division algorithm described below can generate pseudo-quotient bits q_k from any given y in that set. From these, and from a table of precomputed log constants $\log(1+2^{-k})$, we may compute

$$\log(y) = -q_1\log(3/2) - q_2\log(5/4) - \ldots - q_k\log(1+2^{-k}) - \ldots$$

by a process called pseudo-multiplication because its shifts and adds resemble binary integer multiplication. To compute $\log(Y)$ for an arbitrary positive binary floating-point number Y, we can first infer from its exponent field an integer n for which

$$1/\eta(1) < 1/2 \le y = Y/2^n \le 1,$$

and then compute $\log(Y) = n+\log(y)$ by pseudo-division and pseudo-multiplication followed by an addition.

Conversely, if $0 \le x < \log(\eta(1)) = 1.253524...$ then a somewhat different pseudo-division process can decompose x into

$$x = q_1\log(3/2) + q_2\log(5/4) + \ldots + q_k\log(1+2^{-k}) + \ldots$$

with pseudo-quotient bits q_k each either 0 or 1; and then a different pseudo-multiplication process can compute

$$\mathrm{txp}(x) = (1+q_1/2)(1+q_2/4)(1+q_3/8)(\ldots)(1+q_k/2^k)(1+\ldots)$$

(recall $\mathrm{txp}(x) = 2^x$) using just one shift-and-add per factor. And for any floating-point number $X = n+x$, with the integer n so chosen that $0 \le x \le 1 < \log(\eta(1))$, $\mathrm{txp}(X) = 2^n\mathrm{txp}(x)$ can now be computed by pseudo-division, pseudo-multiplication and an addition to the exponent field.

The algorithms for $\log(Y)$ and $\mathrm{txp}(X)$ both access the same table of log-constants $\log(1+2^{-k})$. Of course only finitely many of these can be kept in storage. To compensate for the fact that the last of them is $\log(1+2^{-L})$ for some positive integer L to be determined later, we introduce

$$\begin{aligned} y_L &= y(1+q_1/2)(1+q_2/4)(\ldots)(1+q_L/2^L) \\ &= 1/((1+q_{L+1}/2^{L+1})(1+q_{L+2}/2^{L+2})(1+\ldots)) \end{aligned}$$

and verify that $1/\eta(2^{-L}) < y_L \le 1$. Because y_L lies so close to 1 when L is big enough, $\log(y_L)$ can be approximated well by some rational or polynomial function of y_L perhaps as simple as $y_L - 1$, as we shall see. Therefore

$$\log(Y) = n - \sum_1^L q_k\log(1+2^{-k}) + \log(y_L)$$

can be computed accurately enough from L pseudo-quotient bits and log-constants, and from such an approximation to $\log(y_L)$.

Similarly, we find that

$$0 \le x_L = x - \sum_1^L q_k\log(1+2^{-k}) = \sum_{L+1}^{\infty} q_k\log(1+2^{-k}) < \log(\eta(2^{-L})),$$

so when L is big enough then x_L is tiny enough that $\mathrm{txp}(x_L)$ can be approximated well by some simple rational or polynomial function of x_L that we shall develop later; and then

$$\mathrm{txp}(X) = 2^X = 2^n(1+q_1/2)(1+q_2/4)(...)(1+q_L/2^L)\mathrm{txp}(x_L)$$

can be computed accurately enough from L pseudo-quotient bits and log-constants, and from an approximation to $\mathrm{txp}(x_L)$.

3 Complications to Cope with Cancellation

Roundoff undermines accuracy in two ways, one associated with cancellation and the other with accumulation. We shall deal with the cancellation problem first.

Consider what happens in the formulas for $\log(Y)$ above when Y barely exceeds 1; say $Y = 1 + \xi$ for some very tiny ξ. Then $\log(Y) = \xi/\ln(2)$ very nearly. Now $n = 1$ for $y = Y/2^n$, so $\log(y) = -(1 - \xi/ln(2))$ very nearly. But if ξ is tiny enough, a few orders of magnitude bigger than rounding errors in $\log(y)$, then $\log(y)$ will round to $-(1 - \eta/\ln(2))$ instead, where η matches ξ in at most its first few sig. bits even though those sig. bits lie far to the right of the binary point, and then the value computed for $\log(Y) = 1 + \log(y)$ will be $\eta/\ln(2)$ instead of $\xi/\ln(2)$, differing in all but the first few sig. bits. For instance, pretend arithmetic is rounded to five sig. dec., and suppose that $\log(Y) = 0.00014426$; then $\log(y) = -0.99985574$ rounds to -0.99986 to five sig. dec., and then the computed value of $\log(Y)$ is $1 - 0.99986 = 0.00014000$. As revealed by cancellation, the absolute error is tiny but the relative error is tantamount to losing over half the sig. dec. carried. There are calculations, for example those involving $\log(Y)/(Y - 1)$, where error like that could have disconcerting consequences.

To avoid this error revealed by cancellation, we have to restrict the range of the reduced argument $y = Y/2^n$, keeping it closer to 1. Two ways to do so come to mind. The simplest reduces y initially to an interval that includes the range $1/\sqrt{2} < y < \sqrt{2}$ but not much more. If this reduced $y < 1$, proceed as before; otherwise replace y by $y' = 1/y$, computed actually from the formula $1 - y' = (y - 1)/y$ for reasons to become clear later, and then replace $\log(y)$ by $-\log(y')$. Thus, both y and y' are now restricted to numbers between roughly $1/\sqrt{2}$ and 1, and when Y is close to 1 then n = 0 and severe cancellation need never occur. The Intel 8087 family of numeric coprocessors do this to keep the error in their log functions below two units in the last (64th) bit. But this first simple way to avoid severe cancellation costs an extra division to compute $1 - y'$.

The second way is subtle and requires another table of constants

$$-\log(1 - 2^{-k}),$$

but may be faster because it needs no division by y. This time reduce Y to $y = Y/2^n$ in the range $2/3 < y < 4/3$, and then examine $y - 1$. If nonzero, its floating-point exponent will tell which positive integer M satisfies either

$$-2^{-M} < y - 1 \le -2^{-M-1} \text{ or } 2^{-M-1} \le y - 1 < 2^{-M}.$$

If $M = 1$ here substitute $M = 2$ below. If $y < 1$, pseudo-divide as before except that the calculation of pseudo-quotient bits can start with q_M since every earlier $q_k = 0$. But if $y > 1$ proceed differently: First replace $y - 1$ by

$$y' - 1 = (1 - 2^{-M})y - 1 = (y - 1) - 2^{-M}y;$$

this will be calculated exactly in floating-point arithmetic, and y' will lie between 27/32 and 1, well within the interval from $1/\eta(1/4) = 0.786417...$ to 1 wherein pseudo-division starting from q_3 can succeed. Thereafter do unto $y' - 1$ what would have been done unto $y - 1$; first determine from its floating-point exponent which leading pseudo-quotient bits are predictably zero and so need not be generated, compute $\log(y')$, and finally get

$$\log(y) = -\log(1 - 2^{-M}) + \log(y'),$$

wherupon no more than a bit or two will be lost to cancellation.

A similar pair of tricks is available to ameliorate cancellation during the calculation of $\text{txpm1}(x) = 2^x - 1$ when x is a small negative number. As we shall see later, pseudo-division and pseudo-multiplication can compute $\text{txpm1}(x')$ directly only for some related $x' > 0$. One possibility is $x' = -x$, from which first $t = \text{txpm1}(x')$ and then $\text{txpm1}(x) = -t/(1 + t)$ can be calculated at the cost of one extra division; this is what the Intel 8087 family does. Another possibility is faster when the table of constants $-\log(1 - 2^{-k})$ is available. By comparing the significand of x with $1/\ln(2)$, find the integer M for which $2^{-M-1}/\ln(2) < -x < 2^{-M}/\ln(2)$; set $x' = x - \log(1 - 2^{-M}) > 0$; get $t = \text{txpm1}(x')$ by pseudo-division-and-multiplication; and $\text{txpm1}(x) = t - 2^{-M}(1 + t)$ loses at most two bits to cancellation.

4 Reversing Order Reduces Roundoff's Accumulation

If not by cancellation, another way to lose accuracy to rounding errors is to accumulate too many of them. They do accumulate during the pseudo-division and pseudo-multiplication processes. For instance the sum $\sum_{k=M}^{L} q_k \log(1+2^{-k})$ must suffer one rounding error in each log-constant and one in every addition in that sum. How those rounding errors affect the final sum is determined mostly by the order of summation.

The natural order of summation would form that sum

$$(...(q_M \log(1+2^{-M}) + q_{M+1} \log(1+2^{-M-1})) + ...) + q_L \log(1+2^{-L})$$

simultaneously with the generation of the pseudo-quotient bits, adding the terms in the sum in order of increasing subscripts (from left to right), so each pseudo-quotient bit is consumed as soon as it is generated. This is the fastest way to compute $\log(y)$ but not the most accurate; it loses about $\log(L-M)$ of the bits carried to an accumulation of rounding errors caused by adding $L-M$ ever smaller terms to a slowly changing sum. Just such errors degrade accuracy on the Motorola 68881 and 68882.

Better accuracy during pseudo-multiplication is achieved for the Intel 8087 family by adding the nonzero terms in increasing order of magnitude but decreasing subscripts (right to left) thus:

$$q_M \log(1+2^{-M}) + (q_{M+1} \log(1+2^{-M-1}) + (... + q_L \log(1+2^{-L}))).$$

This process shifts earlier rounding errors off at the right, so the last rounding errors tend to drown out the earlier ones. Now at most one or two bits can be lost to roundoff in the summation process, but the process requires that the pseudo-quotient bits be saved to be consumed in the order opposite to their creation; that is why this pseudo-multiplication process runs moderately slower than the natural order of summation.

Similar considerations arise during the pseudo-multiplication that generates 2^x and $2^x - 1$. For small positive values of x,

$$\text{txpm1}(x) = 2^x - 1 = (1+q_1/2)(1+q_2/4)(...)(1+q_L/2^L)(1+\text{txpm1}(x_L)) - 1$$

can be calculated from left to right, using the pseudo-quotient bits q_k in the order of their creation, or from right to left in reverse order. The latter order is slowed by having to wait until all pseudo-quotient bits have been generated, but that way it accumulates less error. Here is how it works:

Compute $\text{txpm1}(x_L)$ from some approximation valid for nonnegative tiny values $x_L < 2^{-L}/\ln(2)$. Then for $k = L, L-1, L-2, \ldots$ in turn until no nonzero q_k is left unused, calculate

$$\text{txpm1}(x_k) = \text{txpm1}(x_{k+1}) + q_k(1+\text{txpm1}(x_{k+1}))/2^k;$$

finally $\text{txpm1}(x)$ is the last of these calculated. This pseudo-multiplication process shifts earlier rounding errors off at the right so that they do not accumulate beyond a bit or two.

Recurrences like this one for $\text{txpm1}(x_k)$, lying at the heart of pseudo-division and pseudo-multiplication, can be carried out in either floating-point or fixed-point arithmetic. The former is simpler to program; the latter is slightly faster and, when extra bits of precision are available, more accurate too. But the shifts that would be handled automatically in floating-point have to be programmed explicitly in fixed-point lest the accuracy gained by reversing order be lost. The shifts further complicate the algorithms' descriptions.

5 Pseudo-Division in Fixed-Point Arithmetic

The idealized pseudo-division algorithm to compute $\log(y)$ began with a fraction y between $1/\eta(1)$ and 1, and generated

$$\begin{aligned} y_k &= y(1+q_1/2)(1+q_2/4)(1+q_3/8)(1+...)(1+q_k/2^k) \\ &= 1/((1+q_{k+1}/2^{k+1})(1+q_{k+2}/2^{k+2})(1+...)) \ge 1/\eta(2^{-k}) \end{aligned}$$

for $k = 1,2,3,4,...,L$ in turn. Each pseudo-quotient bit q_k is either 0 or 1 according to the following criteria:

If $1/(1+2^{-k}) < y_{k-1} \le 1$ then $q_k = 0$ and $y_k = y_{k-1}$
so $1/\eta(2^{-k}) < 1/(1+2^{-k}) < y_k \le 1$;
if $1/\eta(2^{1-k}) < y_{k-1} < 1/\eta(2^{-k})$ then $q_k = 1$ and $y_k = y_{k-1}(1+2^{-k})$
so $1/\eta(2^{-k}) < y_k < 1$.

These criteria would allow q_k to be chosen arbitrarily as 0 or 1 when $1/\eta(2^{-k}) < y_{k-1} \le 1/(1+2^{-k})$, but the choice has to be algorithmically simple. The following recurrence will do:

If $y_{k-1}(1+2^{-k}) > 1$ then $q_k = 0$ and $y_k = y_{k-1}$
else $q_k = 1$ and $y_k = y_{k-1}(1+2^{-k})$.

This recurrence still poses two problems. One is how to start it. The other is excessive accumulation of roundoff for reasons like those above that motivated reversal of the order of consumption of pseudo-quotient bits during pseudo-multiplication.

To inhibit excessive accumulation of roundoff, we introduce a new scaled variable

$$z_k = 2^k(1 - y_k).$$

We shall choose the first value $k = K$ later. Then the foregoing recurrence takes this form: for $k = K+1, K+2, ..., L$ in turn,

$$z_k = 2z_{k-1} + 2^{1-k}q_k z_{k-1} - q_k \ge 0$$

with $q_k = 1$ unless that would violate the last inequality, in which case $q_k = 0$. Later we shall see why $0 \le z_k < 1$; this permits the z-recurrence to be implemented using a fixed-point register to hold the fractions z_k, a right-shifter to form $2^{-k}z_k$, an adder to add them, and a shift-register to save the pseudo-quotient bits q_k. Rounding errors do occur when bits of $2^{-k}z_k$ are shifted off, but bits shifted off in later stages lie far to the right of bits shifted off in earlier stages, so they do not accumulate. At the end of pseudo-division, the fraction z_L is substituted into a polynomial or rational approximation to $\log(y_L) = \log(1 - 2^{-L}z_L)$ to be chosen later in section mmmm .

How should the z-recurrence start? Recall that pseudo-division is feasible if and only if $1/\eta(2^{-k}) < y_k \le 1$ for every k, and this will be true for every k if it is true for the first. In terms of the scaled variables z_k, the condition necessary and sufficient for successful pseudo-division is $0 \le z_k < \zeta(2^{-k}) < 1$ where $\zeta(t) = (1 - 1/\eta(t))/t$ is discussed with $\eta(t)$ in sect. n, the appendix. These inequalities will be satisfied for all k if satisfied for the first, $k = K$. How shall that be arranged?

Depending upon which of the complications in sect. 2 have been chosen to cope with cancellation, pseudo-division will begin with a floating-point fraction y that lies in an interval like

$$1/\sqrt{2} < y \le 1 \text{ or } 2/3 < y \le 1 \text{ or } 27/32 \le y < 1.$$

That will put the floating-point difference $z = 1 - y$ into one of three corresponding intervals, respectively

$$0 \le z < 1/(2+\sqrt{2}) \text{ or } 0 \le z < 1/3 \text{ or } 0 < z \le 5/32.$$

The integer K that makes $1/4 \le z_K = 2^K z < 1/2$ comes from the floating-point exponent of z. This is the initial value for k in the z_k-recurrence above except for two considerations. First, if $q_K = 0$ advance K to $K+1$ until $q_K = 1$ so that q_K will be the first nonzero pseudo-quotient bit, and record this now to prevent excessive right-shifting later that would lose accuracy. K will be advanced to $K \ge 1$ if z lies in the first two intervals, to $K \ge 2$ if z lies in the third interval above (in which case log-constant $\lambda_1 = \log(3/2)$ will not be needed).

The second aspect of the initial $k = K$ to be considered is what happens in case $K \ge L$ or $z = 0$. In these cases both pseudo-division and pseudo-multiplication can be skipped. $\log(y) = 0$ exactly if $z = 0$; otherwise add 1 to K to get $1/2 \le z_K < 1$ and use the aforementioned approximation to $\log(1 - 2^{-K}z_K)$.

The idealized pseudo-division algorithm to compute $\mathrm{txpm1}(x) = 2^x - 1$ started with a sufficiently small positive number x and used the log-constants $\log(1+2^{-k})$ to generate nonnegative numbers

$$\begin{aligned} x_k &= x - q_1\log(1+2^{-1}) - q_2\log(1+2^{-2}) - ... - q_k\log(1+2^{-k}) \\ &= q_{k+1}\log(1+2^{-k-1}) + q_{k+2}\log(1+2^{-k-2}) + q_{k+2}\log(1+2^{-k-3}) + ... \end{aligned}$$

for $k = 1, 2, 3, 4, ..., L$ in turn. Each pseudo-quotient-bit q_k is either 0 or 1 according to the following criterion:

```
if $x_{k-1} < \log(1+2^{-k})$ then $q_k = 0$ and $x_k = x_{k-1}$
    else $q_k = 1$ and $x_k = x_{k-1} - \log(1+2^{-k})$.
```

This recurrence is numerically satisfactory if carried out in floating-point arithmetic but then it would waste time on exponent comparisons and normalization after cancellation. Faster fixed-point arithmetic requires the introduction of a scaled variable $s_k = 2^{k-1}x_k$ and scaled log-constants $\lambda_k = 2^{k-1}\log(1+2^{-k})$. These log-constants form an increasing sequence of fractions

$$\lambda_1 = 0.58496..., \ \lambda_2 = 0.64385..., \ \lambda_3 = 0.67970..., \ ...$$

bounded above by $\lambda_\infty = 1/\ln(4) = 0.7213475....$ (λ_1 may not be needed.) These scaled log-constants and variable motivate the introduction of analytic functions $\lambda(t) = \log(1+t)/(2t)$ and $\sigma(t) = \log(\eta(t))/(2t)$, both discussed in the appendix, sect n. Apparently every $s_k < \sigma(2^{-k}) < \sigma(0) = \lambda_\infty$ too, so s_k and λ_k can be kept as fractions in fixed-point registers. Fixed-point arithmetic suffices to generate s_k by recurrence:

```
for $k = K+1, K+2, ..., L$ in turn, $s_k = 2s_{k-1} - q_k\lambda_k \geq 0$
```

with $q_k = 1$ unless that would violate the last inequality, in which case $q_k = 0$. Provided the fixed-point registers are wide enough, no rounding errors occur during this recurrence, and rounding errors in later log-constants λ_k fall to the right of bits already rounded off in earlier ones without accumulating.

But how is the initial k = K chosen?

n. Appendix: Properties of $\eta(t)$, $\zeta(t)$, $\sigma(t)$ and $\lambda(t)$

Assertions appearing herein without proof may be unobvious but, if so, they are tedious and routine applications of the calculus.

Definitions:

$\eta(t) = (1+t/2)(1+t/4) * (1+t/8)(1+t/16)(...)(1+t/2^k)(1+...)$

$\zeta(t) = (1-1/\eta(t))/t;\ \zeta(0) = 1.$ $\qquad \eta(t) = 1/(1-t\zeta(t)).$

$\sigma(t) = log(\eta(t))/(2t).$ $\qquad \eta(t) = 2^{2t\sigma(t)}.$

$\lambda(t) = \log(1+t)/(2t) = \ln(1+t)/(t\ln 4).$

Log constants: $\lambda_k = \lambda(2^{-k});\ \mu_k = \lambda(-2^{-k}).$

$\sigma(0) = \lambda(0) = \lambda_\infty = \mu_\infty = 1/\ln(4) = 0.72134752\ldots.$

Pseudo-Quotient Bits: $q_j = q_j^2 = \{0 \text{ or } 1\}.$

Variables: $y = 2^{-x} = 1 - z/2^k,\ x = -\log(y) = 2^{1-k}s.$

Scaled variables: $z = 2^k(1-y),\ s = 2^{k-1}x.$

Conditions for Decompositions:

$1/y = (1+q_{k+1}/2^{k+1})(1+q_{k+2}/2^{k+2})(1+q_{k+3}/2^{k+3})(1+...)$

if and only if $1 \le 1/y \le \eta(2^{-k})$. Equivalently,

$1/(1-z/2^k) = (1+q_{k+1}/2^{k+1})(1+q_{k+2}/2^{k+2})(1+q_{k+3}/2^{k+3})(1+...)$

if and only if $0 \le z \le \zeta(2^{-k})$. Equivalently,

$x = q_{k+1}\log(1+2^{-k-1}) + q_{k+2}\log(1+2^{-k-2}) + q_{k+3}\log(1+2^{-k-3}) + ...$

if and only if $0 \le x \le \log(\eta(2^{-k}))$. Equivalently,

$s = \sum_1^\infty q_{k+j}\lambda_{k+j}/2^j$ if and only if $0 \le s \le \sigma(2^{-k})$.

Functional Equations:

$\eta(0) = 1$ and $\eta(t) = (1+t/2)\eta(t/2).$

$\zeta(t) = (1+\zeta(t/2)/(2+t).$

$\sigma(t) = \log(1+t/2)/(2t) + \sigma(t/2)/2 = (\lambda(t/2)+\sigma(t/2))/2.$

Power Series:

$\eta(t) = ((((((... + 1)t/63 + 1)t/31 + 1)t/15 + 1)t/7 + 1)t/3 + 1)t + 1.$

$\zeta(t)= (((((... - 1)32t/63 + 1)16t/31 - 1)8t/15 + 1)4t/7 - 1)2t/3 + 1$

$= 1/(t + 1/(((((... + 1)t/31 + 1)t/15 + 1)t/7 + 1)t/3 + 1)).$

$\sigma(t) = \sum_1^\infty(-t)^{j-1}/((2^j-1)j\ln 4).$ $\lambda(t) = (1 - t/2 + t^2/3 - t^3/4 + t^4/5 - ...)/\ln(4).$

Inequalities and Monotonicity Properties:

For all $t \ge -2, \eta'(t) > 0$;

for all nonzero $t \ge -4$, $\eta''(t) > 0$ and $1 + t < \eta(t) < e^t$.

For$0 < t \le 1, \zeta'(t) < 0 < \zeta''(t)$ and

$1 - 14t/(21+12t) < \zeta(t) < 1 - 26t/(39+23t) < 1 = \zeta(0).$

(The rational expressions bracket $\zeta(t)$ within ±0.005.)

For $0 \le t \le 1$, $\sigma'(t) < 0 < \sigma''(t)$.

For $-1 < t \le 1$, $\lambda'(t) < 0 < \lambda''(t)$.

A Short Table:

k	$\eta(-2^{-k})$	$\eta(2^{-k})$	$\zeta(2^{-k})$	$\sigma(2^{-k})$	λ_k	μ_k
0	0.288788	2.384231	0.580578	0.626762	0.5	∞
1	0.577576	1.589487	0.741733	0.668562	0.584963	1
2	0.770102	1.271590	0.854332	0.693267	0.643856	0.830075
3	0.880116	1.130302	0.922246	0.706834	0.679700	0.770580
4	0.938791	1.063814	0.959773	0.713965	0.699703	0.744875
5	0.969074	1.031577	0.979533	0.717624	0.710306	0.732859
6	0.984456	1.015707	0.989676	0.719477	0.715770	0.727042
∞	1	1	1	0.721348	0.721348	0.721348

Miscellaneous Properties:

All but six of the entries in the table above have to be rounded off because they appear not to be representable exactly as fixed- or floating-point numbers. Are appearances deceptive? It is not hard to confirm that $\eta(\pm1)$ must be irrational, whence the same soon follows for $\eta(\pm2^{-k})$ and $\zeta(2^{-k})$ when $k < \infty$. Similarly for all λ_k and μ_k except λ_0, μ_0 and μ_1. I don't know about $\sigma(2^{-k})$ yet.

Pseudo-division in this paper constrains the pseudo-quotient bits q_j to the set $\{0, 1\}$. Other sets work too, but not so neatly. For example, if $q_j = -q_j^2 = \{0 \text{ or } -1\}$ then

$$1/y = (1 + q_{k+1}/2^{k+1})(1 + q_{k+2}/2^{k+2})(1 + q_{k+3}/2^{k+3})(1 + ...)$$

must lie in the interval $\eta(-2^{-k}) \le 1/y \le 1$; but some values $1/y$ in that interval, for instance those in $1 - 2^{-k-1} < 1/y < \eta(-2^{-k-1})$, cannot be decomposed into the factorization shown above wherein no power of $1/2$ appears more than once. For another example, if $q_k^2 = 1$ (so $q_k = \pm1$) then $1/y$ must lie in the interval $\eta(-2^{-k}) \le 1/y \le \eta(2^{-k})$; but some values $1/y$ in that interval cannot be decomposed without repeating some powers of $1/2$. An instance between $\eta(-1)$ and $\eta(1)$ is $1/y = 0.48$ because it has

$$\begin{aligned} 1/((1-1/2)(1-1/4)y) &= 1.28 > \eta(1/4) \text{ and} \\ 1/((1-1/2)(1+1/4)y) &= 0.768 < \eta(-1/4), \end{aligned}$$

so no factorization with every q_j restricted to ±1 can exist. Therefore, any advantage that might be secured by using a set other than $\{0, 1\}$ for the pseudo-quotient bits q_j must be purchased at the cost of a pseudo-division scheme more complicated than is described in this paper.