Computing Logarithms

F. Auger, L. Zhen, B. Feuvrie

IREENA, Université de Nantes, France.

E-mail: {francois.auger, bruno.feuvrie}@univ-nantes.fr, luozhen.lz@126.com

THE BASIC PRINCIPLE

To compute the logarithm of a positive number x, the principle of this method is to find the mathematical inverse of x, i.e., the number y such that

$$x \cdot y = 1. \tag{1}$$

Taking the logarithm of both sides of Eq. (1) yields

$$\log(x) + \log(y) = 0$$

or

$$\log(x) = -\log(y). \tag{2}$$

The number y in Eq. (2) is expanded so that it is much easier to compute $\log(y)$ than it is to compute $\log(x)$. Once we compute $\log(y)$ we negate that value to obtain $\log(x)$. Based on this simple basic principle (which is of course not an algorithm) an algorithm can be designed. Our logarithm algorithm is multiplier free and iterative, and therefore is a little more complicated than this basic principle.

THE ALGORITHM

In our algorithm the value y in Eq. (1) takes the form:

$$y = (1 + 2^{-0})^{k_0} (1 + 2^{-1})^{k_1} (1 + 2^{-2})^{k_2} (1 + 2^{-3})^{k_3} \ldots (1 + 2^{-N})^{k_N} \tag{3}$$

where the k_i sequence of exponents are integers, defined such that

- $x \cdot 2^{k_0}$ is between 0.5 and 1,
- k_1 is the highest integer such that $x \cdot 2^{k_0}(1 + 2^{-1})^{k_1}$ is less than 1 (which means that $x \cdot 2^{k_0}(1 + 2^{-1})^{(k_1+1)}$ is greater than 1,

- k_2 is the highest integer such that $x{\cdot}2^{k_0}(1 + 2^{-1})^{k_1}(1 + 2^{-2})^{k_2}$ is less than 1 (which means that $x{\cdot}2^{k_0}(1 + 2^{-1})^{k_1}(1 + 2^{-2})^{(k_2+1)}$ is greater than 1,
- and so on …

As such, the product $x{\cdot}y$ becomes

$$x{\cdot}y = x\,[2^{k_0}(1 + 2^{-1})^{k_1}(1 + 2^{-2})^{k_2}(1 + 2^{-3})^{k_3} \ldots (1 + 2^{-N})^{k_N}] \approx 1. \qquad (4)$$

Taking the logarithm, to the base b, of Eq. (4) gives us

$$\log_b(x{\cdot}y) = \log_b(x) + k_0 \log_b(2) + k_1 \log_b(1 + 2^{-1}) + k_2 \log_b(1 + 2^{-2})$$

$$+ k_3 \log_b(1 + 2^{-3})^{k_3} \ldots + k_N \log_b(1 + 2^{-N}) \approx 0. \qquad (5)$$

which implies

$$\log_b(x) \approx -\sum_{i=0}^{i=N} k_i \log_b(1 + 2^{-i}) \qquad (6)$$

Our final algorithm, then, will compute the series of products given in Eq. (6). However, while the absolute value of integer k_0 can be greater than one, the remaining k_1 through k_N integers in Eq. (6) can only be zeros or ones. For example, for $x = 5$, $k_0 = -3$ and all the following k_i are equal to zero except k_1, k_4 , k_8, k_{16} , k_{32} … which are all equal to 1. This allows us to avoid performing multiplications when computing the right side of Eq. (6).

We completely eliminate multiplications by first precomputing the sequence $\log_b(1 + 2^{-i})$, for $0 \le i \le N$, and storing those constants in a lookup table (LUT). If, for example, $k_0 = -3$ we compute the first term in Eq. (6) by accumulating three of the $-\log_b(2)$ constants from the LUT. To that accumulated value, we add the remaining $\log_b(1 + 2^{-i})$ LUT constants based on whether k_i is a zero or a one. This way no multiplications are needed.

The value N in Eq. (6) is user-defined based on the desired precision of our $\log_b(x)$ result. The larger N, more terms will be accumulated in Eq. (6), and the more precise will be the computed $\log_b(x)$. It can hence be shown that

$$\frac{2^N}{2^N+1} < x{\cdot}y \le 1,$$

which shows that for a large N, the product $x{\cdot}y$ is very close to 1.

COMPUTING THE k_i FACTORS

In an iterative implementation of our algorithm, shown in Figure 1, we don't actually compute the k_i factors explicitly. Here's what we do to compute $\log_b(x)$. Assuming that x is a positive number, we initialize an accumulator, log$x$ in Figure 1, to zero. Next, if x is greater that one, we perform binary right shifts until the shifted x is less than one. For each right shift we add $\log_b(2)$ to the *logx* accumulator. If, on the other hand, the original x was less than 0.5 we perform left shifts until the shifted x is greater than or equal to 0.5. For each left shift we subtract $\log_b(2)$ from the *logx* accumulator. Those additions (or subtractions) of $\log_b(2)$ complete the computation of the $-k_0\log_b(2)$ term in Eq. (6).

Now that the shifted x is between 0.5 and 1, we iteratively subtract, where appropriate (i.e. when we can add $2^{-i}{\cdot}x$ to x without exceeding 1), the remaining LUT $\log_b(1 + 2^{-i})$ terms from the *logx* accumulator until i reaches N and the accumulated *logx* reaches the desired accuracy. (The value N is represented by variable N_{bits}, the desired number of bits in the $\log_b(x)$ result, in Figure 1.)

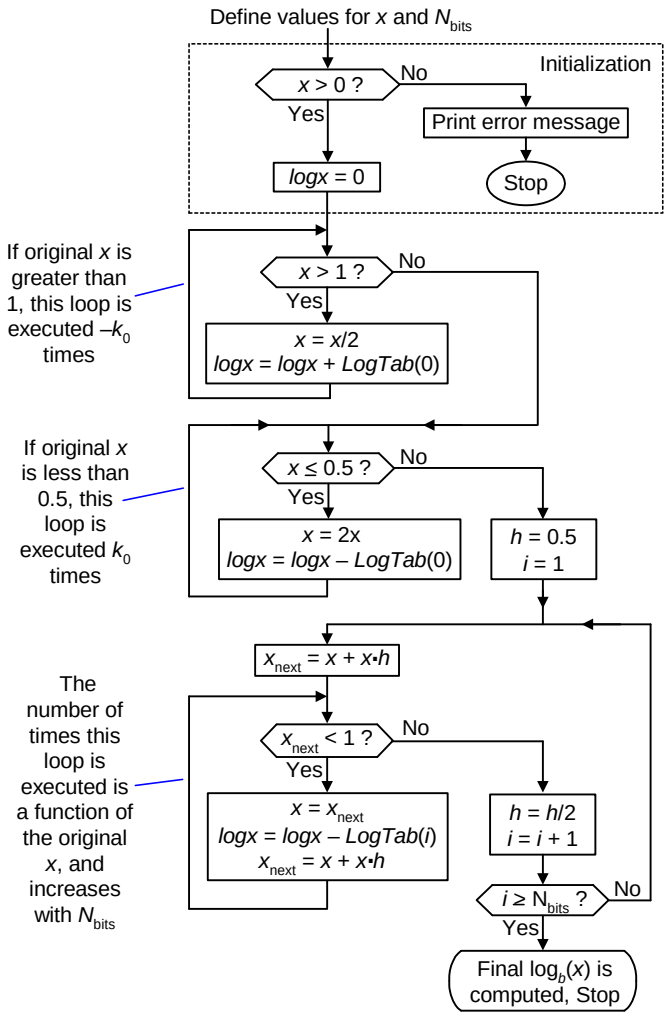

FIG. 1 Computational flow diagram of the logarithm algorithm.