Computing Integer Powers in Floating-Point Arithmetic

Peter Kornerup Vincent Lefèvre Jean-Michel Muller [*]

May 2007
This is LIP Research Report number RR2007-23
Ceci est le Rapport de Recherches numéro RR2007-23 du LIP
Laboratoire LIP, CNRS/ENS Lyon/INRIA/Univ. Lyon 1, Lyon, France.

Abstract

We introduce two algorithms for accurately evaluating powers to a positive integer in floating-point arithmetic, assuming a *fused multiply-add* (fma) instruction is available. We show that our log-time algorithm always produce faithfully-rounded results, discuss the possibility of getting correctly rounded results, and show that results correctly rounded in double precision can be obtained if extended-precision is available with the possibility to round into double precision (with a single rounding).

1 Introduction

We deal with the implementation of the integer power function in floating-point arithmetic. In the following, we assume a radix-2 floating-point arithmetic that follows the IEEE-754 standard for floating-point arithmetic. We also assume that a fused multiply-and-add (fma) operation is available, and that the input as well as the output values of the power function are not subnormal numbers, and are below the overflow threshold (so that we can focus on the powering of the significands only).

An important case dealt with in the paper will be the case when an internal format, wider than the target format, is available. For instance, to guarantee – in some cases – correctly rounded integer powers in double precision arithmetic, we will have to assume that a double-extended precision is available. The examples will consider that it has a 64-bit precision, which is the minimum required by the IEEE-754 standard.

The IEEE-754 standard [1] for radix-2 floating-point arithmetic (and its follower, the IEEE-854 radix-independent standard [5]) require that the four arithmetic operations and the square root should be correctly rounded. In a floating-point system that follows the standard, the user can choose an *active rounding mode* from:

- rounding towards $-\infty$: $RD(x)$ is the largest machine number less than or equal to x;

[*]Peter Kornerup is with SDU, Odense, Denmark; Vincent Lefèvre and Jean-Michel Muller are with Laboratoire LIP, CNRS/ENS Lyon/INRIA/Univ. Lyon 1, Lyon, France.

- rounding towards $+\infty$: $RU(x)$ is the smallest machine number greater than or equal to x;
- rounding towards 0: $RZ(x)$ is equal to $RD(x)$ if $x \geq 0$, and to $RU(x)$ if $x < 0$;
- rounding to nearest: $RN(x)$ is the machine number that is the closest to x (with a special convention if x is exactly between two machine numbers: the chosen number is the "even" one, i.e., the one whose last significand bit is a zero).

When $a \circ b$ is computed, where a and b are floating-point numbers and $\circ$ is $+$, $-$, $\times$ or $\div$, the returned result is what we would get if we computed $a \circ b$ exactly, with "infinite" precision and rounded it according to the active rounding mode. The default rounding mode is round-to-nearest. This requirement is called *correct rounding.* Among its many interesting properties, one can cite the following result (the first ideas that underlie it go back to Møller [10]).

Theorem 1 (Fast2Sum algorithm) *(Theorem C of [6], page 236). Assume the radix r of the floating-point system being considered is less than or equal to* 3*, and that the used arithmetic provides correct rounding with rounding to nearest. Let a and b be floating-point numbers, and assume that the exponent of a is larger than or equal to that of b. The following algorithm computes two floating-point numbers s and t that satisfy:*

- *$s + t = a + b$ exactly;*
- *s is the floating-point number that is closest to $a + b$.*

Algorithm 1 (Fast2Sum(a,b))

$$\begin{array}{rcl} s & := & RN(a+b); \\ z & := & RN(s-a); \\ t & := & RN(b-z); \end{array}$$

If no information on the relative orders of magnitude of a and b is available, there is an alternative algorithm introduced by Knuth [6]. It requires 6 operations instead of 3 for the Fast2Sum algorithm, but on any modern computer, the 3 additional operations cost significantly less than a comparison followed by a branching.

Some processors (e.g., the IBM PowerPC or the Intel/HP Itanium [2]) have a *fused multiply-add* (fma) instruction that allows to compute $ax \pm b$, where a, x and b are floating-point numbers, with one final rounding only. This instruction allows one to design convenient software algorithms for correctly rounded division and square root. It also has the following interesting property. From two input floating-point numbers a and b, the following algorithm computes c and d such that $c+d = ab$, and c is the floating-point number that is nearest ab.

Algorithm 2 (Fast2Mult(a,b))

```
$c$ := $RN\,(ab)$;
$d$ := $RN\,(ab - c)$;
```

Performing a similar calculation without a fused multiply-add operation is possible [3] but requires 17 floating-point operations instead of 2.

Algorithms Fast2Sum and Fast2Mult both provide double-precision results of value $(x + y)$ represented in the form of pairs (x, y). In the following we need product of numbers represented in this form. However, we will be satisfied with approximations to the product, discarding terms of the order of the product of the two low-order terms. Given two double-precision operands $(a_h + a_l)$ and $(b_h + b_l)$ the following algorithm *DblMult* computes (x, y) such that $(x + y) = [(a_h + a_l)(b_h + b_l)](1 + \delta)$ where the relative error δ is discussed in Section 3 below.

Algorithm 3 (DblMult(a_h,a_l,b_h,b_l))

```
$t$          := $RN\,(a_l b_h)$;
$s$          := $RN\,(a_h b_l + t)$;
$(x', u)$    := $Fast2Mult\,(a_h, b_h)$;
$(x'', v)$   := $Fast2Sum\,(x', s)$;
$y'$         := $RN\,(u + v)$;
$(x, y)$     := $Fast2Sum\,(x'', y')$;
```

Note that the condition for applying $Fast2Sum$ is satisfied.

2 The two algorithms

We now give two algorithms for accurately computing x^n, where x is a floating-point number, and n is an integer greater than or equal to 1. We assume that an fma instruction is available, as it is used in $Fast2Mult$ and thus implicitly also in *DblMult*.

The first ($\mathcal{O}(n)$ time) algorithm is derived from the straightforward, $(n - 1)$-multiplication, algorithm. It is simple to analyze and will be faster than the other one if n is small.

Algorithm 4 (LinPower(x, n), $n \geq 1$)

```
$(h, l) := (x, 0)$;
for $i$ from $2$ to $n$ do
   $(h, v) := Fast2Mult\,(h, x)$;
         $l := RN\,(l\,x + v)$;
end do;
return $(h, l)$;
```

where the low order terms are accumulated with appropriate weights using a Horner scheme evaluation. Algorithm LinPower uses $3n - 3$ floating-point operations.

The second ($\mathcal{O}(\log(n))$-time) algorithm is based on successive squarings.

Algorithm 5 (LogPower(x, n), $n \geq 1$)

```
$i := n$;
$(h, l) := (1, 0)$;
$(u, v) := (x, 0)$;
while $i > 1$ do
  if ($i$ mod 2) $= 1$ then
    $(h, l) :=$ DblMult$(h, l, u, v)$;
  end;
  $(u, v) :=$ DblMult$(u, v, u, v)$;
  $i := \lfloor i/2 \rfloor$;
end do;
return DblMult$(h, l, u, v)$;
```

Due to the approximations performed in algorithm *DblMult*, terms corresponding to the product of low order terms are not included. A thorough error analysis is performed below. The number of floating-point operations used by the LogPower algorithm is between $11(1+\lfloor \log_2(n) \rfloor)$ and $11(1+2\lfloor \log_2(n) \rfloor)$, whereas for LinPower it is $3(n-1)$. Hence, LogPower will become faster than LinPower for values of n around 30 (but counting the floating-point operations only gives a rough estimate, the actual threshold will depend on the architecture and compiler).

3 Error analysis

We will use the following result.

Theorem 2 (Theorem 2.2 of [4], p. 38) *Assume a radix-r floating-point system F, with precision p. If $x \in \mathbb{R}$ lies in the range of F, then*

$$RN(x) = x(1+\delta), \ |\delta| < \frac{1}{2} r^{-p+1}.$$

3.1 Error of function DblMult

Theorem 3 *Let $\epsilon = 2^{-p}$, where p is the precision of the radix-2 floating-point system used. If $|a_l| \leq 2^{-p}|a_h|$ and $|b_l| \leq 2^{-p}|b_h|$ then the returned value (x, y) of function DblMult(a_h, a_l, b_h, b_l) satisfies*

$$x + y = (a_h + a_l)(b_h + b_l)(1 + \eta),$$

with

$$|\eta| \leq 6\epsilon^2 + 16\epsilon^3 + 17\epsilon^4 + 11\epsilon^5 + 5\epsilon^6 + \epsilon^7.$$

Notes:

1. as soon as $p \geq 5$, we have $|\eta| \leq 7\epsilon^2$;
2. in the case of single precision ($p = 24$), $|\eta| \leq 6.000001\epsilon^2$;
3. in the case of double precision ($p = 53$), $|\eta| \leq (6 + 2 \times 10^{-15})\,\epsilon^2$.

Proof: Following the notation in Algorithm 5, with ϵ_i's being variables of absolute value less than ϵ, we have

$$\begin{aligned}
x+y &= x'' + RN\,(u+v)\\
&= x'' + (u+v)(1+\epsilon_1)\\
&= (x''+v) + u + u\epsilon_1 + v\epsilon_1\\
&= x' + s + u + u\epsilon_1 + v\epsilon_1\\
&= a_h b_h + s + u\epsilon_1 + v\epsilon_1\\
&= a_h b_h + [a_h b_l + (a_l b_h)(1+\epsilon_3)](1+\epsilon_2) + u\epsilon_1 + v\epsilon_1\\
&= a_h b_h + a_h b_l + a_l b_h + a_h b_l \epsilon_2 + a_l b_h \epsilon_2 + a_l b_h \epsilon_2\epsilon_3 + a_l b_h \epsilon_3 + u\epsilon_1 + v\epsilon_1.
\end{aligned}$$

We also have $a_l = \epsilon_4 a_h$, $b_l = \epsilon_5 b_h$, $u = \epsilon_6 a_h b_h$, and

$$\begin{aligned}
v &= \epsilon_7(x'+s)\\
&= \epsilon_7\left(a_h b_h(1+\epsilon_8) + [a_h b_l + a_l b_h(1+\epsilon_3)](1+\epsilon_2)\right)\\
&= \epsilon_7\left(a_h b_h(1+\epsilon_8) + [\epsilon_5 a_h b_h + \epsilon_4 a_h b_h(1+\epsilon_3)](1+\epsilon_2)\right)\\
&= \epsilon_7 a_h b_h\left(1+\epsilon_8+\epsilon_5+\epsilon_2\epsilon_5+\epsilon_4+\epsilon_2\epsilon_4+\epsilon_3\epsilon_4+\epsilon_2\epsilon_3\epsilon_4\right)\\
&= \eta_1 a_h b_h,
\end{aligned}$$

with $|\eta_1| \le \epsilon + 3\epsilon^2 + 3\epsilon^3 + \epsilon^4$. Hence

$$\begin{aligned}
x+y &= a_h b_h + a_h b_l + a_l b_h + (a_l b_l - \epsilon_4\epsilon_5 a_h b_h) + a_h b_h(\epsilon_2\epsilon_5 + \epsilon_2\epsilon_4 + \epsilon_2\epsilon_3\epsilon_4 + \epsilon_3\epsilon_4 + \epsilon_1\epsilon_6 + \eta_1\epsilon_1)\\
&= (a_h+a_l)(b_h+b_l) + a_h b_h \eta_2,
\end{aligned}$$

with $|\eta_2| \le 6\epsilon^2 + 4\epsilon^3 + 3\epsilon^4 + \epsilon^5$.

Now, from $a_h = (a_h+a_l)(1+\epsilon_9)$ and $b_h = (b_h+b_l)(1+\epsilon_{10})$ we deduce

$$x+y = (a_h+a_l)(b_h+b_l)(1+\eta),$$

with $\eta = (1+\epsilon)^2\eta_2$, which gives $|\eta| \le 6\epsilon^2 + 16\epsilon^3 + 17\epsilon^4 + 11\epsilon^5 + 5\epsilon^6 + \epsilon^7$. □

3.2 Error of algorithm LogPower

Theorem 4 *The two values h and l returned by algorithm LogPower satisfy*

$$h + l = x^n(1+\alpha),$$

with

$$(1-|\eta|)^{n-1} \le 1+\alpha \le (1+|\eta|)^{n-1}$$

where $|\eta| \le 6\epsilon^2 + 16\epsilon^3 + 17\epsilon^4 + 11\epsilon^5 + 5\epsilon^6 + \epsilon^7$ is the same value as in Theorem 3.

Proof: Algorithm LogPower computes approximations to powers of x, using $x^{i+j} = x^i x^j$. By induction, one easily shows that the approximation to x^k is of the form $x^k(1+\beta_k)$, where $(1-|\eta|)^{k-1} \le (1+\beta_k) \le (1+|\eta|)^{k-1}$. If we call η_{i+j} the relative error (obtained from Theorem 3) when multiplying together the approximations to x^i and x^j, the induction follows from

$$(1-\eta)^{i-1}(1-\eta)^{j-1}(1-\eta) \le \left(x^i(1+\beta_i)\right)\left(x^j(1+\beta_j)\right)(1+\eta_{i+j}) \le (1+\eta)^{i-1}(1+\eta)^{j-1}(1+\eta).$$

□

Table 1 gives bounds on $|\alpha|$ for several values of n (note that the bound is an increasing value of n), assuming the algorithm is used in double precision.

Define the *significand* of a non-zero real number u to be

$$\frac{u}{2^{\lfloor \log_2 |u| \rfloor}}.$$

Define α_{max} as the bound on $|\alpha|$ obtained for a given value of n. From

$$x^n(1-\alpha_{max}) \leq h+l \leq x^n(1+\alpha_{max}),$$

we deduce that the significand of $h+l$ is within $2\alpha_{max}$ from $x^n/2^{\lfloor \log_2 |h+l| \rfloor}$. From the results given in Table 1, we deduce that for all practical values of n the significand of $h+l$ is within much less than 2^{-53} from $x^n/2^{\lfloor \log_2 |h+l| \rfloor}$ (indeed, to get $2\alpha_{\max}$ larger that 2^{-53}, we need $n > 2^{49}$). This means that $RN(h+l)$ is within less than one ulp from x^n, hence

Theorem 5 *If algorithm LogPower is implemented in double precision, then $RN(h+l)$ is a faithful rounding of x^n, as long as $n \leq 2^{49}$.*

n	$-\log_2(\alpha_{max})$	n	$-\log_2(\alpha_{max})$
3	102.41	1000	93.45
4	101.83	10,000	90.12
5	101.41	100,000	86.80
10	100.24	1,000,000	83.48
20	99.16	10,000,000	80.16
30	98.55	100,000,000	76.83
40	98.12	2^{32}	71.41
50	97.80		
100	96.78		
200	95.77		

Table 1: Binary logarithm of the relative accuracy ($-\log_2(\alpha_{max})$), for various values of n assuming algorithm LogPower is used in double precision.

Moreover, for $n \leq 10^8$, $RN(h+l)$ is within 0.50000007 ulps from the exact value: we are very close to correct rounding (indeed, we almost always return a correctly rounded result), yet we cannot guarantee correct rounding, even for the smallest values of n. This requires a much better accuracy, as shown in Section 4. To guarantee a correctly rounded result in double precision, we will need to run algorithm LogPower in double-extended precision. Table 2 gives bounds on $|\alpha|$ for several values of n assuming the algorithm is realized in double-extended precision. As expected, we are 22 bits more accurate.

3.3 Error of algorithm LinPower

Define h_i, v_i, l_i as the values of variables h, v and l at the end of the loop of index i of the algorithm. Define $\hat{l}_i$ as the value variable l_i would have if the instructions $l := RN(lx+v)$ were errorless (that is, if instead we had $l := (lx+v)$ exactly):

$$\hat{l}_i = v_i + v_{i-1}x + v_{i-2}x^2 + v_{i-3}x^3 + \cdots + v_2x^{i-2}. \tag{1}$$

n	$-\log_2(\alpha_{max})$	n	$-\log_2(\alpha_{max})$
3	124.41	1000	115.45
4	123.83	10,000	112.12
5	123.41	100,000	108.80
10	122.24	1,000,000	105.48
20	121.16	10,000,000	102.16
30	120.55	100,000,000	98.83
40	120.12	2^{32}	93.41
50	119.80		
100	118.78		
200	117.77		

Table 2: Binary logarithm of the relative accuracy ($-\log_2(\alpha_{max})$), for various values of n assuming algorithm LogPower is implemented in double-extended precision.

Initially let $h_1 = x$, $v_1 = l_1 = 0$. By induction, one can easily show that

$$x^i = h_i + v_i + v_{i-1}x + v_{i-2}x^2 + v_{i-3}x^3 + \cdots + v_2 x^{i-2}, \tag{2}$$

hence we have

$$x^i = h_i + \hat{l}_i.$$

The algorithm only computes an approximation l_i to $\hat{l}_i$. To evaluate the error of the algorithm, we must therefore estimate the distance between l_i and $\hat{l}_i$. We have $l_1 = \hat{l}_1 = 0$, and $l_2 = \hat{l}_2 = v_2$ exactly. Define ϵ_i as the number of absolute value less than $\epsilon = 2^{-p}$ such that

$$l_i = RN\,(l_{i-1}x + v_i) = (l_{i-1}x + v_i)(1 + \epsilon_i).$$

We have $l_3 = \hat{l}_3(1 + \epsilon_3)$, and by induction, we find for $i \geq 4$, using $v_i = \hat{l}_i - \hat{l}_{i-1}x$:

$$\begin{array}{rcl} l_i & = & \hat{l}_i(1+\epsilon_i) \\ & + & \hat{l}_{i-1}\epsilon_{i-1}x(1+\epsilon_i) \\ & + & \hat{l}_{i-2}\epsilon_{i-2}x^2(1+\epsilon_{i-1})(1+\epsilon_i) \\ & \vdots & \\ & + & \hat{l}_3\epsilon_3 x^{i-3}(1+\epsilon_4)(1+\epsilon_5)\cdots(1+\epsilon_{i-1})(1+\epsilon_i). \end{array} \tag{3}$$

To derive a useful bound from this result, we must make a simplifying hypothesis. We know that $|v_i| \leq \epsilon|h_i|$. We assume h_i is close enough to x^i, so that

$$|v_i| \leq 2\epsilon|x|^i$$

(this means that our estimate for x^n will become wrong when the algorithm becomes very inaccurate for x^i, $i \leq n$). From (1), we therefore have:

$$|\hat{l}_i| \leq 2(i-1)\epsilon|x|^i,$$

from which, using (3), we deduce

$$l_n = \hat{l}_n + \eta,$$

where

$$|\eta| \leq 2|x|^n\epsilon^2\left[(n-1)+(n-2)(1+\epsilon)+(n-3)(1+\epsilon)^2+\cdots+2(1+\epsilon)^{n-3}\right]. \quad (4)$$

This gives the following result

Theorem 6 (Accuracy of algorithm LinPower) *If for $i<n, |v_i| \leq 2^{1-p}|x|^i$, the final computed values h_n and l_n of the variables h and l of the algorithm satisfy*

$$h_n + l_n = x^n(1+\alpha),$$

where $|\alpha| \leq 2\epsilon^2\left[(n-1)+(n-2)(1+\epsilon)+(n-3)(1+\epsilon)^2+\cdots+2(1+\epsilon)^{n-3}\right].$

Let us try to compute an estimate of the coefficient $\gamma = (n-1)+(n-2)(1+\epsilon)+(n-3)(1+\epsilon)^2+\cdots+2(1+\epsilon)^{n-3}$ in α.

Define a function

$$\varphi(t) = t^{n-1}+(1+\epsilon)t^{n-2}+(1+\epsilon)^2t^{n-3}+\cdots+(1+\epsilon)^{n-3}t^2.$$

One can notice that $\gamma = \varphi'(1)$, so that if we are able to find a simple formula for $\varphi(t)$ we will be able to deduce a formula for γ. We have

$$\varphi(t) = (1+\epsilon)^{n-1}\left[\left(\frac{t}{1+\epsilon}\right)^{n-1}+\left(\frac{t}{1+\epsilon}\right)^{n-2}+\cdots+\left(\frac{t}{1+\epsilon}\right)^{2}\right],$$

hence

$$\varphi(t) = (1+\epsilon)^{n-1}\left[\frac{\left(\frac{t}{1+\epsilon}\right)^n-1}{\frac{t}{1+\epsilon}-1}-\frac{t}{1+\epsilon}-1\right].$$

Thus

$$\varphi'(t) = (1+\epsilon)^{n-2}\left[\frac{(n-1)\left(\frac{t}{1+\epsilon}\right)^n-n\left(\frac{t}{1+\epsilon}\right)^{n-1}+1}{\left(\frac{t}{1+\epsilon}-1\right)^2}-1\right],$$

Hence a bound on the value of $|\alpha|$ is,

$$|\alpha| \leq 2\epsilon^2(1+\epsilon)^{n-2}\left[\frac{(n-1)\left(\frac{1}{1+\epsilon}\right)^n-n\left(\frac{1}{1+\epsilon}\right)^{n-1}+1}{\left(\frac{1}{1+\epsilon}-1\right)^2}-1\right] \approx (n^2-n-2)\epsilon^2.$$

Table 3 gives the obtained bound on $|\alpha|$ for several values of n, assuming double precision ($\epsilon = 2^{-53}$). That table shows that as soon as n is larger than a few units, algorithm LinPower is less accurate than algorithm LogPower.

4 Correct rounding

In this section we consider algorithm LogPower only: first because it is the fastest for all reasonable values of n, second because it is the only one for which we have certain error bounds (the error bounds of algorithm LinPower are approximate only). And if needed, specific algorithms could be designed for small values of n. We are interested

n	$-\log_2(\alpha_{max})$
3	104.00
4	102.68
5	101.83
10	99.54
20	97.43
30	96.23
100	92.72

Table 3: Binary logarithm of the relative accuracy ($-\log_2(\alpha_{max})$), for various values of n assuming algorithm LinPower is implemented in double precision.

in getting correctly rounded results in double precision. To do so, we assume that we perform algorithm LogPower in double extended precision. The algorithm returns two double-extended numbers h and l such that

$$x^n(1-\alpha_{max}) \leq h + l \leq x^n(1+\alpha_{max}),$$

where α_{max} is given in Table 2.

In the following we will need to distinguish two roundings, e.g., RN_e means round-to-nearest in extended double precision and RN_d is round-to-nearest in double precision. Let ulp($\cdot$) denote "unit-in-last-position" such that $|x - RN(x)| \leq \frac{1}{2}\mathrm{ulp}(x)$.

V. Lefèvre introduced a new method for finding hardest-to-round cases for evaluating a regular function [8, 7]. That method allowed Lefèvre and Muller to give such cases for the most familiar elementary functions [9]. Recently, Lefèvre adapted his method to the case of functions x^n and $x^{1/n}$, when n is an integer. For instance, in double-precision arithmetic, the hardest to round case for function x^{51} corresponds to

$$x = 1.0100010111101011011011101010011111100101000111011101$$

we have

$$\begin{aligned} x^{51} &= \underbrace{1.1011001110100100011100100001100100000101101011101110}_{53 \text{ bits}}\ 1 \\ &\quad \underbrace{0000000000\cdots0000000000}_{59 \text{ zeros}}100\cdots \times 2^{17} \end{aligned}$$

which means that x^n is extremely close to the exact middle of two consecutive double-precision numbers. There is a run of 59 consecutive zeros after the rounding bit. This case is the worst case for all values of n between 3 and 145. Table 4 gives the maximal length of the chains of identical bits after the rounding bit for $3 \leq n \leq 145$.

n	Number of identical bits after the rounding bit
32	48
$76, 81, 85$	49
$9, 15, 16, 31, 37, 47, 54, 55, 63, 65, 74, 80, 83, 86, 105, 109, 126, 130$	50
$10, 14, 17, 19, 20, 23, 25, 33, 34, 36, 39, 40, 43, 46, 52, 53, 72, 73, 75, 78, 79, 82, 88, 90, 95, 99, 104, 110, 113, 115, 117, 118, 119, 123, 125, 129, 132, 133, 136, 140$	51
$3, 5, 7, 8, 22, 26, 27, 29, 38, 42, 45, 48, 57, 60, 62, 64, 68, 69, 71, 77, 92, 93, 94, 96, 98, 108, 111, 116, 120, 121, 124, 127, 128, 131, 134, 139, 141$	52
$6, 12, 13, 21, 58, 59, 61, 66, 70, 102, 107, 112, 114, 137, 138, 145$	53
$4, 18, 44, 49, 50, 97, 100, 101, 103, 142$	54
$24, 28, 30, 41, 56, 67, 87, 122, 135, 143$	55
$89, 106$	56
$11, 84, 91$	57
$35, 144$	58
51	59

Table 4: Maximal length of the chains of identical bits after the rounding bit (assuming the target precision is double precision) in the worst cases for n from 3 to 145.

Define a *breakpoint* as the exact middle of two consecutive double precision numbers. $RN_d\,(h+l)$ will be equal to $RN_d\,(x^n)$ if and only if there is no breakpoint between x^n and $h+l$.

The worst case obtained shows that if x is a double-precision number, and if $3 \leq n \leq 145$, then the significand y of x^{51} is always at a distance larger than 2^{-113} from the breakpoint μ (see Figure 1) where the distance $|y-\mu| \geq 2^{-(53+59+1)} = 2^{-113}$.

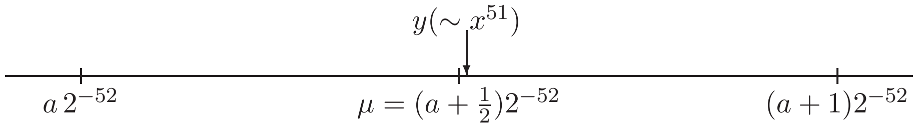

Figure 1: Position of the hardest to round case $y = x^{51}$ within rounding interval $[a2^{-52}; (a+1)2^{-52}]$ with breakpoint $\mu = (a+\frac{1}{2})2^{-52}$, for significand defined by integer a.

We know that the significand of $h+l$ is within $2\alpha_{max}$ from that of x^n, where α_{max} (as given by its binary logarithm) is listed in Table 2. For all values of n less than or equal to 145, we have $2\alpha_{max} \leq 2^{-113}$, thus $RN_d\,(h+l) = RN_d\,(x^n)$. We therefore get the following result:

Theorem 7 *If algorithm LogPower is run in double-extended precision, and if* $3 \leq n \leq 145$*, then* $RN_d\,(h+l) = RN_d\,(x^n)$*: Hence by rounding h+l to the nearest double-precision number, we get a correctly rounded result.*

Now, two important remarks:

- We do not have the worst cases for $n > 145$, but from probabilistic arguments we strongly believe that the lengths of the largest chains of consecutive bits after the rounding bit will be of the same order of magnitude (i.e., around 50) for some range of n above 145. However, it is unlikely that we will be able to show correct rounding in double precision for values of n larger than 1000.

- On an Intel Itanium processor, it is possible to directly add two double-extended precision numbers and round the result to double precision without a "double rounding" (i.e., without having an intermediate sum rounded to double-extended precision). Hence Theorem 7 can directly be used. It is worth being noticed that the draft revised standard IEEE 754-R (see `http://754r.ucbtest.org/`) includes the fma as well as rounding to any specific destination format, independent of operand formats.

Conclusion

It has been shown that the function x^n can be calculated in time $\mathcal{O}(\log n)$ with correct rounding in double precision, employing double-extended precision arithmetic, at least for the range $3 \leq n \leq 145$. A fused multiply accumulate (fma) instruction is assumed available for algorithm efficiency reasons; and to keep the analysis simple, it was assumed that the input as well as the output are not subnormal numbers and are below the overflow threshold.

A simpler, $\mathcal{O}(n)$ time algorithm, faster than the above for small values of n, was also analyzed. However, its error analysis turned out to be more complicated (and less rigorous), and also to be less accurate than the other.

References

[1] American National Standards Institute and Institute of Electrical and Electronic Engineers. IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard, Std 754-1985,* New York, 1985.

[2] M. Cornea, J. Harrison, and P. T. P. Tang. *Scientific Computing on Itanium-Based Systems.* Intel Press, Hillsboro, OR, 2002.

[3] T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 3 1971.

[4] N. Higham. *Accuracy and Stability of Numerical Algorithms, Second Edition.* SIAM, Philadelphia, PA, 2002.

[5] American National Standards Institute, Institute of Electrical, and Electronic Engineers. IEEE standard for radix independent floating-point arithmetic. *ANSI/IEEE Standard, Std 854-1987, New York*, 1987.

[6] D. Knuth. *The Art of Computer Programming, 3rd edition*, volume 2. Addison-Wesley, Reading, MA, 1998.

[7] V. Lefèvre. *Developments in Reliable Computing*, chapter An Algorithm That Computes a Lower Bound on the Distance Between a Segment and Z^2, pages 203–212. Kluwer Academic Publishers, Dordrecht, 1999.

[8] V. Lefèvre. *Moyens Arithmétiques Pour un Calcul Fiable.* PhD thesis, École Normale Supérieure de Lyon, Lyon, France, 2000.

[9] V. Lefèvre and J.-M. Muller. Worst cases for correct rounding of the elementary functions in double precision. In Burgess and Ciminiera, editors, *Proc. of the 15th IEEE Symposium on Computer Arithmetic (Arith-15).* IEEE Computer Society Press, Los Alamitos, CA, 2001.

[10] 0. Møller. Quasi double-precision in floating-point addition. *BIT*, 5:37–50, 1965.