Computing Correctly Rounded Integer Powers in Floating-Point Arithmetic

PETER KORNERUP
Southern Danish University, Odense
and
CHRISTOPH LAUTER, VINCENT LEFÈVRE, NICOLAS LOUVET,
and JEAN-MICHEL MULLER
ENS Lyon

We introduce several algorithms for accurately evaluating powers to a positive integer in floating-point arithmetic, assuming a *fused multiply-add* (fma) instruction is available. For bounded, yet very large values of the exponent, we aim at obtaining correctly rounded results in round-to-nearest mode, that is, our algorithms return the floating-point number that is nearest the exact value.

Categories and Subject Descriptors: G.1.0 [**Numerical Analysis**]: General—*Computer arithmetic, error analysis*; G.4 [**Mathematical Software**]: Algorithm design and analysis, reliability and robustness

General Terms: Algorithms, Reliability, Performance

Additional Key Words and Phrases: Correct rounding, floating-point arithmetic, integer power function

ACM Reference Format:

Kornerup, P., Lauter, C., Lefèvre, V., Louvet, N., and Muller, J.-M. 2010. Computing correctly rounded integer powers in floating-point arithmetic. ACM Trans. Math. Softw. 37, 1, Article 4 (January 2010), 23 pages.
DOI = 10.1145/1644001.1644005 http://doi.acm.org/10.1145/1644001.1644005

This work was partly supported by the French Agence Nationale de la Recherche (ANR), through the EVA-Flo project.

Author's current addresses: P. Kornerup, Department of Mathematics & Computer Science, Southern Danish University, Odense, Campusvej 55, DK-5230 Odense M, Denmark; email: kornerup@imada.sdu.dk; C. Lauter, Intel Corporation, Software and Services Group, 2111 NE 25th Avenue, M/S JF1-13, Hillsboro, OR 97124; email: christoph.lauter@intel.com; V. Lefèvre, INRIA, Laboratoire LIP, ENS Lyon, 46 Allèe d'Italie, 69364 Lyon cedex 07, France; email: vincent@vinc17.net; N. Louvet, Université Lyon 1, Laboratoire LIP, ENS Lyon, 46 Allée d'Italie, 69364 Lyon cedex 07, France; email: Nicolas.Louvet@ens-lyons.fr; J.-M. Muller, CNRS, Laboratoire LIP, ENS Lyon, 46 Allèe d'Italie, 69364 Lyon cedex 07, France; email: jean-michel.muller@ens-lyons.fr.

DOI 10.1145/1644001.1644005 http://doi.acm.org/10.1145/1644001.1644005

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 fma instruction allows one to compute $ax \pm b$, where a, x, and b are floating-point numbers, with one final rounding only. Examples of processors with an fma are the IBM PowerPC and the Intel/HP Itanium [Cornea et al. 2002].

An important case dealt with in the article is 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 using our algorithms based on iterated products, we will have to assume that an extended precision[1] is available. The examples will assume that it has a 64-bit precision, which is the minimum required by the IEEE-754 standard.

The only example of currently available processor with an fma and an extended-precision format is Intel and HP's Itanium Processor [Li et al. 2002; Cornea et al. 2002]. And yet, since the fma operation is now required in the revised version of the IEEE-754 standard [IEEE 2008], it is very likely that more processors in the future will offer that combination of features.

The original IEEE-754 standard [American National Standards Institute and Institute of Electrical and Electronic Engineers 1985] for radix-2 floating-point arithmetic as well as its follower, the IEEE-854 radix-independent standard [American National Standards Institute and Institute of Electrical and Electronic Engineers 1987], and the new revised standard [IEEE 2008] 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;

—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 halfway between two consecutive machine numbers: the chosen number is the "even" one, that is, the one whose last significand bit is a zero).[2]

[1]It was called "double extended" in the original IEEE-754 standard.

[2]The new version of the IEEE 754-2008 standard also specifies a "Round ties to Away" mode. We do not consider it here.

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 (due to Dekker [1971]).

THEOREM 1 (FAST2SUM ALGORITHM). *Assume the radix r of the floating-point system being considered is 2 or 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

function $[s, t] = FastSum(a, b)$
$s := RN(a + b)$;
$z := RN(s - a)$;
$t := RN(b - z)$;

Note that the assumption that the exponent of a is larger than or equal to that of b is ensured if $|a| \geq |b|$, which is sufficient for our proofs in the sequel of the paper. In the case of radix 2, it has been proven in Rump et al. [2005-2008] that the result still holds under the weaker assumption that a is an integer multiple of the unit in the last place of b.

If no information on the relative orders of magnitude of a and b is available a priori, then a comparison and a branch instruction are necessary in the code to appropriately order a and b before calling Fast2Sum (algorithm 1), which is costly on most microprocessors. On current pipelined architectures, a wrong branch prediction causes the instruction pipeline to drain. Moreover, both conditions "the exponent of a is larger or equal to that of b" and "a is a multiple of the unit in the last place of b" require access to the bit patterns of a and b, which is also costly and results in a poorly portable code. However, there is an alternative algorithm due to Knuth [1998] and Møller [1965], called 2Sum (algorithm 2). It requires six operations instead of three for the Fast2Sum algorithm, but on any modern computer the three additional operations cost significantly less than a comparison followed by a branching.

The fma 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, Algorithm 3 computes c and d such that $c + d = ab$, and c is the floating-point number that is nearest ab.

Algorithm 2. 2SUM

function $[s, t] = 2Sum(a, b)$
$s := RN(a + b)$;
$b' := RN(s - a)$;
$a' := RN(s - b')$;
$\delta_b := RN(b - b')$;
$\delta_a := RN(a - a')$;
$t := RN(\delta_a + \delta_b)$;

Performing a similar calculation without a fused multiply-add operation is possible with an algorithm due to Dekker [1971], but this would require 17 floating-point operations instead of two. Transformations such as 2Sum, Fast2Sum, and Fast2Mult were called *error-free transformations* by Rump [Ogita et al. 2005].

In the sequel of the article, we examine various methods for getting very accurate (indeed: correctly rounded, in round-to-nearest mode) integer powers. We first present in Section 2 our results about the hardest-to-round cases for the power function x^n. At the time of writing, we have determined these worst cases only for n up to 733: as a consequence, we will always assume $3 \leq n \leq 733$ in this article. In Section 3, we deal with methods based on repeated multiplications, where the arithmetic operations are performed with a larger accuracy using algorithms such as Fast2Sum and Fast2Mult. We then investigate in Section 4 methods based on the identity

$$x^n = 2^{n \log_2(x)},$$

using techniques we have developed when building the CRlibm library of correctly rounded mathematical functions [Daramy-Loirat et al. 2006; de Dinechin et al. 2007]. We report in Section 5 timings and comparisons for the various proposed methods. In Section 6, we present our conclusions.

2. ON CORRECT ROUNDING OF FUNCTIONS

V. Lefèvre [1999; 2000; 2005] introduced a new method for finding hardest-to-round cases for evaluating a regular unary function. That method allowed Lefèvre and Muller [2001] to give such cases for the most familiar elementary functions. Recently, Lefèvre adapted his software to the case of functions x^n, where n is an integer; this consisted in supporting a parameter n.

Let us briefly summarize Lefèvre's method. The tested domain is split into intervals, where the function can be approximated by a polynomial of a quite small degree and an accuracy of about 90 bits. The approximation does not need

Algorithm 3. FAST2MULT

function $[c, d] = Fast2Mult(a, b)$
$c := RN(ab)$;
$d := RN(ab - c)$;

to be very tight, but it must be computed quickly; that is why Taylor's expansion is used. For instance, by choosing intervals of length 1/8192 of a binade, the degree is 3 for $n = 3$, it is 9 for $n = 70$, and it is 12 to 13 for $n = 500$. These intervals are split into subintervals where the polynomial can be approximated by polynomials of smaller degrees, down to 1. How intervals are split exactly depends on the value of n (the parameters can be chosen empirically, thanks to timing measures). Determining the approximation for the following subinterval can be done using fixed-point additions in a precision up to a few hundreds of bits, and multiplications by constants. A filter of sublinear complexity is applied on each degree-1 polynomial, eliminating most input arguments. The remaining arguments (e.g., one over 2^{32}) are potential worst cases, which need to be checked in a second step by direct computation in a higher accuracy.

Because of a reduced number of input arguments, the second step is much faster than the first step and can be run on a single machine. The first step (every computation up to the filter) is parallelized. The intervals are independent, so that the following conventional solution has been chosen: a server distributes intervals to the clients running on a small network of desktop machines.

All approximation errors are carefully bounded, either by interval arithmetic or by static analysis. Additional checks for missing or corrupt data are also done in various places. So, the final results are guaranteed (up to undetected software bugs and errors in paper proofs).[3]

Concerning the particular case of x^n, one has $(2x)^n = 2^n x^n$. Therefore if two numbers x and y have the same significand, their images x^n and y^n also have the same significand. So only one binade needs to be tested,[4] $[1, 2)$ in practice.

For instance, in double-precision arithmetic, the hardest to round case for function x^{458} corresponds to

$$x = 1.0000111100111000110011111010101011001011011100011010,$$

for which we have

$$x^{458} = \underbrace{1.00011111000010110000100001110110101110100000000100101}_{53 \text{ bits}}\ 1$$
$$\underbrace{00000000\cdots00000000}_{61 \text{ zeros}}1110\cdots \times 2^{38},$$

which means that x^n is extremely close to the exact middle of two consecutive double-precision numbers. There is a run of 61 consecutive zeros after the rounding bit. This case is the worst case for all values of n between 3 and 733.

This worst case has been obtained by an exhaustive search using the method described above, after a total of 646,300 hours of computation for the first step (sum of the times on each CPU core) on a network of processors. The time needed to test a function x^n increases with n, as the error on the approximation by a degree-1 polynomial on some fixed interval increases. On the current network

[3]Work has started toward formalized proofs (in the EVA-Flo project, funded by the French Agence Nationale de la Recherche).

[4]We did not take subnormals into account, but one can prove that the worst cases in all rounding modes can also be used to round subnormals correctly.

(when all machines are available), for $n \simeq 600$, it takes between 7 and 8 h for each power function. On a reference 2.2-Ghz AMD Opteron machine, one needs an estimated time of 90 h core to test x^n with $n = 10$, about 280 h for $n = 40$, and around 500 h for any n between 200 and 600.

Table I gives the longest runs of identical bits after the rounding bit for $3 \le n \le 733$.

3. ALGORITHMS BASED ON REPEATED MULTIPLICATIONS

3.1 Using a Double-Double Multiplication Algorithm

Algorithms Fast2Sum and Fast2Mult both provide "double-FP" results, namely, couples (a_h, a_ℓ) of floating-point numbers such that (a_h, a_ℓ) represents $a_h + a_\ell$ and satisfies $|a_\ell| \le \frac{1}{2}\mathrm{ulp}(a_h)$. In the following, we need products of numbers represented in this form. However, we will be satisfied with approximations to the products, discarding terms of the order of the product of the two low-order terms. Given two double-FP operands $(a_h + a_\ell)$ and $(b_h + b_\ell)$, the following algorithm DblMult (Algorithm 3) computes (c_h, c_ℓ) such that $(c_h + c_\ell) = [(a_h + a_\ell)(b_h + b_\ell)](1 + \eta)$, where the relative error η is given by Theorem 2 below.

Algorithm 4. DblMult

function $[c, d] = DblMult(a_h, a_\ell, b_h, b_\ell)$
  $[t_{1h}, t_{1\ell}] := FastMult(a_h, b_h)$;
  $t_2 := RN(a_h b_\ell)$;
  $t_3 := RN(a_\ell b_h + t_2)$;
  $t_4 := RN(t_{1\ell} + t_3)$;
  $[c_h, c_\ell] := FastSum(t_{1h}, t_4)$;

THEOREM 2. *Let $\varepsilon = 2^{-p}$, where $p \ge 3$ is the precision of the radix-2 floating-point system used. If $|a_\ell| \le 2^{-p}|a_h|$ and $|b_\ell| \le 2^{-p}|b_h|$ then the returned value $[c_h, c_\ell]$ of function $DblMult(a_h, a_\ell, b_h, b_\ell)$ satisfies*

$$c_h + c_\ell = (a_h + a_\ell)(b_h + b_\ell)(1 + \alpha), \quad with \quad |\alpha| \le \eta,$$

where $\eta := 7\varepsilon^2 + 18\varepsilon^3 + 16\varepsilon^4 + 6\varepsilon^5 + \varepsilon^6$.

Notes:

(1) as soon as $p \ge 5$, we have $\eta \le 8\varepsilon^2$;
(2) in the case of single precision ($p = 24$), $\eta \le 7.000002\varepsilon^2$;
(3) in the case of double precision ($p = 53$), $\eta \le 7.000000000000002\varepsilon^2$;
(4) in the case of extended precision ($p \ge 64$), $\eta \le 7.00000000000000000002\varepsilon^2$.

Moreover $DblMult(1, 0, u, v) = DblMult(u, v, 1, 0) = [u, v]$, that is, the multiplication by [1, 0] is exact.

PROOF. We assume that $p \ge 3$. Exhaustive tests in a limited exponent range showed that the result still holds for $p = 2$, and the proof could be completed for untested cases in this precision, but this is out of the scope of this article.

Table I. Maximal Length k of the Runs of Identical Bits After the Rounding Bit (Assuming the Target Precision is Double Precision) in the Worst Cases for n from 3 to 733

n	k
32	48
76, 81, 85, 200, 259, 314, 330, 381, 456, 481, 514, 584, 598, 668	49
9, 15, 16, 31, 37, 47, 54, 55, 63, 65, 74, 80, 83, 86, 105, 109, 126, 130, 148, 156, 165, 168, 172, 179, 180, 195, 213, 214, 218, 222, 242, 255, 257, 276, 303, 306, 317, 318, 319, 325, 329, 342, 345, 346, 353, 358, 362, 364, 377, 383, 384, 403, 408, 417, 429, 433, 436, 440, 441, 446, 452, 457, 459, 464, 491, 494, 500, 513, 522, 524, 538, 541, 547, 589, 592, 611, 618, 637, 646, 647, 655, 660, 661, 663, 673, 678, 681, 682, 683, 692, 698, 703, 704	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, 146, 149, 150, 155, 157, 158, 162, 166, 170, 174, 185, 188, 189, 192, 193, 197, 199, 201, 205, 209, 210, 211, 212, 224, 232, 235, 238, 239, 240, 241, 246, 251, 258, 260, 262, 265, 267, 272, 283, 286, 293, 295, 296, 301, 302, 308, 309, 324, 334, 335, 343, 347, 352, 356, 357, 359, 363, 365, 371, 372, 385, 390, 399, 406, 411, 412, 413, 420, 423, 431, 432, 445, 447, 450, 462, 465, 467, 468, 470, 477, 482, 483, 487, 490, 496, 510, 518, 527, 528, 530, 534, 543, 546, 548, 550, 554, 557, 565, 567, 569, 570, 580, 582, 585, 586, 591, 594, 600, 605, 607, 609, 610, 615, 616, 622, 624, 629, 638, 642, 651, 657, 665, 666, 669, 671, 672, 676, 680, 688, 690, 694, 696, 706, 707, 724, 725, 726, 730	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, 152, 154, 161, 163, 164, 173, 175, 181, 182, 183, 184, 186, 196, 202, 206, 207, 215, 216, 217, 219, 220, 221, 223, 225, 227, 229, 245, 253, 256, 263, 266, 271, 277, 288, 290, 291, 292, 294, 298, 299, 305, 307, 321, 322, 323, 326, 332, 349, 351, 354, 366, 367, 369, 370, 373, 375, 378, 379, 380, 382, 392, 397, 398, 404, 414, 416, 430, 437, 438, 443, 448, 461, 471, 474, 475, 484, 485, 486, 489, 492, 498, 505, 507, 508, 519, 525, 537, 540, 544, 551, 552, 553, 556, 563, 564, 568, 572, 575, 583, 593, 595, 597, 601, 603, 613, 619, 620, 625, 627, 630, 631, 633, 636, 640, 641, 648, 650, 652, 654, 662, 667, 670, 679, 684, 686, 687, 702, 705, 709, 710, 716, 720, 721, 727	52
6, 12, 13, 21, 58, 59, 61, 66, 70, 102, 107, 112, 114, 137, 138, 145, 151, 153, 169, 176, 177, 194, 198, 204, 228, 243, 244, 249, 250, 261, 268, 275, 280, 281, 285, 297, 313, 320, 331, 333, 340, 341, 344, 350, 361, 368, 386, 387, 395, 401, 405, 409, 415, 418, 419, 421, 425, 426, 427, 442, 449, 453, 454, 466, 472, 473, 478, 480, 488, 493, 499, 502, 506, 509, 517, 520, 523, 526, 532, 533, 542, 545, 555, 561, 562, 571, 574, 588, 590, 604, 608, 614, 621, 626, 632, 634, 639, 644, 653, 658, 659, 664, 677, 689, 701, 708, 712, 714, 717, 719	53
4, 18, 44, 49, 50, 97, 100, 101, 103, 142, 167, 178, 187, 191, 203, 226, 230, 231, 236, 273, 282, 284, 287, 304, 310, 311, 312, 328, 338, 355, 374, 388, 389, 391, 393, 394, 400, 422, 428, 434, 435, 439, 444, 455, 469, 501, 504, 511, 529, 535, 536, 549, 558, 559, 560, 566, 573, 577, 578, 581, 587, 596, 606, 612, 623, 628, 635, 643, 649, 656, 675, 691, 699, 700, 711, 713, 715, 718, 731, 732	54
24, 28, 30, 41, 56, 67, 87, 122, 135, 143, 147, 159, 160, 190, 208, 248, 252, 264, 269, 270, 279, 289, 300, 315, 339, 376, 396, 402, 410, 460, 479, 497, 515, 516, 521, 539, 579, 599, 602, 617, 674, 685, 693, 723, 729	55
89, 106, 171, 247, 254, 278, 316, 327, 348, 360, 424, 451, 463, 476, 495, 512, 531, 645, 697, 722, 728	56
11, 84, 91, 234, 237, 274, 407, 576, 695	57
35, 144, 233, 337, 733	58
51, 336	59
503	60
458	61

We will prove that the exponent of t_4 is less than or equal to the exponent of t_{1h}. Thus we have $t_{1h} + t_{1\ell} = a_h b_h$ and $c_h + c_\ell = t_{1h} + t_4$ (both exactly).

Now, let us analyze the other operations. In the following, the ε_i's are terms of absolute value less than or equal to $\varepsilon = 2^{-p}$. First, notice that $a_\ell = \varepsilon_1 a_h$ and $b_\ell = \varepsilon_2 b_h$. Since the additions and multiplications are correctly rounded (to nearest) operations:

(1) $t_2 = a_h b_\ell(1 + \varepsilon_3)$;

(2) $t_{1\ell} = a_h b_h \varepsilon_4$ since $(t_{1h}, t_{1\ell}) = \mathit{FastMult}(a_h, b_h)$;

(3) $t_3 = (a_\ell b_h + t_2)(1 + \varepsilon_5)$
$= a_h b_\ell + a_\ell b_h + a_h b_h(\varepsilon_1\varepsilon_5 + \varepsilon_2\varepsilon_3 + \varepsilon_2\varepsilon_5 + \varepsilon_2\varepsilon_3\varepsilon_5)$
$= a_h b_\ell + a_\ell b_h + a_h b_h(3\varepsilon_6^2 + \varepsilon_6^3)$;

(4) $t_4 = (t_{1\ell} + t_3)(1 + \varepsilon_7)$
$= t_{1\ell} + a_h b_\ell + a_\ell b_h + a_h b_h(\varepsilon_4\varepsilon_7 + \varepsilon_2\varepsilon_7 + \varepsilon_1\varepsilon_7 + (3\varepsilon_6^2 + \varepsilon_6^3)(1 + \varepsilon_7))$
$= t_{1\ell} + a_h b_\ell + a_\ell b_h + a_h b_h(6\varepsilon_8^2 + 4\varepsilon_8^3 + \varepsilon_8^4)$.
We also need:
$t_4 = a_h b_h(\varepsilon_4 + \varepsilon_2 + \varepsilon_1 + 6\varepsilon_8^2 + 4\varepsilon_8^3 + \varepsilon_8^4) = a_h b_h(3\varepsilon_9 + 6\varepsilon_9^2 + 4\varepsilon_9^3 + \varepsilon_9^4)$,
where

$$\left|3\varepsilon_9 + 6\varepsilon_9^2 + 4\varepsilon_9^3 + \varepsilon_9^4\right| \leq 1$$

for $p \geq 3$. Thus $|t_4| \leq |a_h b_h|$ and the exponent of t_4 is less than or equal to the exponent of t_{1h}. From that we deduce the following:

(5) $c_h + c_\ell = t_{1h} + t_4$
$= a_h b_h + a_h b_\ell + a_\ell b_h + a_h b_h(6\varepsilon_8^2 + 4\varepsilon_8^3 + \varepsilon_8^4)$
$= a_h b_h + a_h b_\ell + a_\ell b_h + a_\ell b_\ell + a_h b_h(6\varepsilon_8^2 + 4\varepsilon_8^3 + \varepsilon_8^4 - \varepsilon_1\varepsilon_2)$
$= (a_h + a_\ell)(b_h + b_\ell) + a_h b_h(7\varepsilon_{10}^2 + 4\varepsilon_{10}^3 + \varepsilon_{10}^4)$.
Now, from $a_h = (a_h + a_\ell)(1 + \varepsilon_{11})$ and $b_h = (b_h + b_\ell)(1 + \varepsilon_{12})$, we get

$$a_h b_h = (a_h + a_\ell)(b_h + b_\ell)(1 + \varepsilon_{11} + \varepsilon_{12} + \varepsilon_{11}\varepsilon_{12}),$$

from which we deduce

$$c_h + c_\ell = (a_h + a_\ell)(b_h + b_\ell)(1 + 7\varepsilon_{13}^2 + 18\varepsilon_{13}^3 + 16\varepsilon_{13}^4 + 6\varepsilon_{13}^5 + \varepsilon_{13}^6). \quad \square$$

3.2 The IteratedProductPower Algorithm

Algorithm 5 below is our algorithm to compute an approximation to x^n (for $n \geq 1$) using repeated multiplications with DblMult. The number of floating-point operations used by the IteratedProductPower algorithm is between $8(1 + \lfloor \log_2(n) \rfloor)$ and $8(1 + 2\lfloor \log_2(n) \rfloor)$.

Algorithm 5. IteratedProductPower

```
function [h, ℓ] = IteratedProductPower(x, n)
  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 := ⌊i/2⌋;
  end do;
  [h, ℓ] := 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.

3.3 Error of Algorithm IteratedProductPower

THEOREM 3. *The values h and ℓ returned by algorithm IteratedProductPower satisfy*

$$h + \ell = x^n(1+\alpha), \quad \textit{with} \quad |\alpha| \le (1+\eta)^{n-1} - 1,$$

where $\eta = 7\varepsilon^2 + 18\varepsilon^3 + 16\varepsilon^4 + 6\varepsilon^5 + \varepsilon^6$ is the same value as in Theorem 2.

PROOF. Algorithm 3, IteratedProductPower, computes approximations to powers of x, using $x^{i+j} = x^i x^j$. By induction, one easily shows that, for $k \ge 1$, the approximation to x^k is of the form $x^k(1+\alpha_k)$, where $|\alpha_k| \le (1+\eta)^{k-1} - 1$. Indeed, the multiplication by $[1, 0]$ is exact, and if we call β the relative error (whose absolute value is bounded by η according to Theorem 2) when multiplying together the approximations to x^i and x^j for $i \ge 1$ and $j \ge 1$, the induction follows from

$$x^i x^j (1+\alpha_i)(1+\alpha_j)(1+\beta) = x^{i+j}(1+\alpha_{i+j})$$

and

$$\begin{aligned} |\alpha_{i+j}| &= |(1+\alpha_i)(1+\alpha_j)(1+\beta) - 1| \le (1+|\alpha_i|)(1+|\alpha_j|)(1+|\beta|) - 1 \\ &\le (1+\eta)^{i+j-1} - 1. \end{aligned}$$

□

Let $\alpha_{\max} := (1+\eta)^{n-1} - 1$ be the upper bound on the accuracy of the approximation to x^n computed by IteratedProductPower. Note that the bound is an increasing value of n. Table II gives lower bounds on $-\log_2(\alpha_{\max})$ for several values of n.

Define the *significand* of a nonzero real number u to be $u/2^{\lfloor \log_2 |u| \rfloor}$. From $h + \ell = x^n(1+\alpha)$, with $|\alpha| \le \alpha_{\max}$, we deduce that $(h+\ell)/2^{\lfloor \log_2 |x^n| \rfloor}$ is within $2\alpha_{\max}$ from the significand of x^n. From the results given in Table II, we deduce that, for all practical values of n, $(h+\ell)/2^{\lfloor \log_2 |x^n| \rfloor}$ is within much less than 2^{-54} from the significand of x^n (indeed, to get $2\alpha_{\max}$ larger than 2^{-54}, we need

Table II. Binary Logarithm of the Relative Accuracy $-\log_2(\alpha_{\max})$, for Various Values of n, Assuming Algorithm IteratedProductPower Is Used in Double Precision

n	$-\log_2(\alpha_{\max})$
3	102.19
10	100.02
100	96.56
1000	93.22
10000	89.90

Table III. Binary Logarithm of the Relative Accuracy $-\log_2(\alpha_{\max})$, for Various Values of n, Assuming Algorithm IteratedProductPower Is Implemented in Extended Precision

n	$-\log_2(\alpha_{\max})$
3	124.19
10	122.02
100	118.56
458	116.35
733	115.67
1000	115.22
10000	111.90

$n > 2^{48}$). This means that $RN(h + \ell)$ is within less than one ulp from x^n, or, more precisely:

THEOREM 4. *If algorithm IteratedProductPower is implemented in double precision, then $RN(h + \ell)$ is a faithful rounding of x^n, as long as $n \leq 2^{48}$.*

Moreover, for $n \leq 10^8$, $RN(h + \ell)$ is within 0.50000008 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 3.4. To guarantee a correctly rounded result in double precision, we will need to run algorithm IteratedProductPower in extended precision.

3.4 Getting Correctly Rounded Values with IteratedProductPower

We are interested in getting correctly rounded results in double precision. To achieve this, we assume that algorithm IteratedProductPower is executed in extended precision. The algorithm returns two extended-precision numbers h and ℓ such that $h + \ell = x^n(1 + \alpha)$, with $|\alpha| \leq \alpha_{\max}$. Table III gives lower bounds on $-\log_2(\alpha_{\max})$ for several values of n, assuming the algorithm is realized in extended precision. As expected, we are 22 bits more accurate. In the following, double-FP numbers based on the extended-precision formats will be called *extended-extended*.

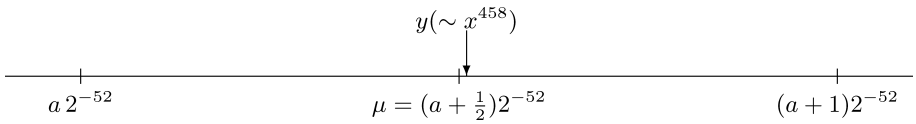

Fig. 1. Position of the hardest to round case $y = x^{458}$ within rounding interval $[a2^{-52}; (a+1)2^{-52}]$ with breakpoint $\mu = (a + \frac{1}{2})2^{-52}$, for significand defined by integer a.

In the following, we shall distinguish two roundings: RNe means round-to-nearest in extended precision and RNd is round-to-nearest in double precision.

Define a *breakpoint* as the exact midpoint of two consecutive double-precision numbers. $RNd(h + \ell)$ will be equal to $RNd(x^n)$ if and only if there are no breakpoints between x^n and $h + \ell$.

For $n \leq 733$, Table I shows that, in the worst case, a run of 61 zeros or ones follows the rounding bit in the binary expansion of x^n, this worst case being obtained for $n = 458$. As a consequence, the significand of x^n is always at a distance larger than $2^{-(53+61+1)} = 2^{-115}$ from the breakpoint μ (see Figure 1). On the other hand, we know that $(h + \ell)/2^{\lfloor \log_2(x^n) \rfloor}$ is within $2\alpha_{\max}$ of x^n, with bounds on $\alpha_{\max}$ given in Table III: the best bound we can state for all values of n less than or equal to 733 is $2\alpha_{\max} \leq 2^{-114.67}$. Therefore, we have to distinguish two cases to conclude about correct rounding:

—if $n = 458$ then, from Table III, we have $2\alpha_{\max} \leq 2^{-115.35}$;
—if $n \neq 458$ then, from the worst cases reported in Table I, we know that the significand x^n is always at a distance larger than $2^{-(53+60+1)} = 2^{-114}$ from the breakpoint μ. Moreover, in this case, $2\alpha_{\max} \leq 2^{-114.67}$.

As a consequence, in both cases $RNd(h + \ell) = RNd(x^n)$, which means that the following result holds.

THEOREM 5. *If algorithm IteratedProductPower is performed in extended precision, and if* $3 \leq n \leq 733$*, then* $RNd(h + \ell) = RNd(x^n)$*: hence by rounding* $h + \ell$ *to the nearest double-precision number, we get a correctly rounded result.*

Now, two important remarks:

—We do not have the worst cases for $n > 733$, but from probabilistic arguments, we strongly believe that the lengths of the largest runs of consecutive bits after the rounding bit will be of the same order of magnitude for some range of n above 733. However, it is unlikely that we will be able to show correct rounding in double precision using the same techniques as above for larger values of n.
—On an Intel Itanium processor, it is possible to directly add two extended-precision numbers and round the result to double precision without a "double rounding" (i.e., without having an intermediate sum rounded to extended precision). Hence Theorem 5 can directly be used. Notice that the revised standard IEEE Std. 754^{tm}-2008 [IEEE 2008] includes the fma as well as rounding to any specific destination format, independent of operand formats.

Table IV. Binary Logarithm of the Relative Accuracy $-\log_2 \alpha_{\max}$ in Theorem 6

n	$-\log_2(\alpha_{\max})$
3	62.99
10	60.83
32	59.04
100	57.37
512	55.00
513	54.99
1000	54.03
10000	50.71

3.5 Two-Step Algorithm Using Extended Precision

Now we suggest another approach: first compute an approximation to x^n using extended precision and a straightforward, very fast, algorithm (Algorithm 6). Then check if this approximation suffices to get $RN(x^n)$. If it does not, use the IteratedProductPower algorithm presented above.

Let us first give the algorithm. All operations are done in extended precision.

Algorithm 6. DBLEXTENDEDPOWER

```
function pow = DbleXtendedPower(x, n)
  i := n;
  pow := 1;
  u := x;
  while i > 1 do
    if (i mod 2) = 1 then
      pow := RN_e(pow · u);
    end;
    u := RN_e(u · u);
    i := ⌊i/2⌋;
  end do;
  pow := RN_e(pow · u);
```

Using the very same proof as for Theorem 3, one easily shows the following result.

THEOREM 6. *The final result* pow *of algorithm DbleXtendedPower satisfies*

$$pow = x^n(1+\alpha), \quad with \quad |\alpha| \le \alpha_{\max},$$

where $\alpha_{\max} = (1+2^{-64})^{n-1} - 1$.

Table IV could be used to show that, up to $n = 512$, Algorithm 4 (run in extended precision) can be used to guarantee faithful rounding (in double precision).

But what is of interest here is correct rounding. For example, if $n \le 32$ then from Table IV, it follows that $|\alpha| < 2^{-59}$, which means that the final result *pow* of algorithm DbleXtendedPower is within $2^{53} \times 2^{-59} = 1/64$ ulp from x^n. This means that, if the bits 54 to 59 of *pow* are not 100000 or 011111, then

rounding *pow* to the nearest floating-point number will be equivalent to rounding x^n. Otherwise, if the bits 54 to 59 of *pow* are 100000 or 011111 (which may occur with probability 1/32), we will have to run a more accurate but slower algorithm, such as algorithm IteratedProductPower. We implemented this approach to check whether correct rounding is already possible from the evaluation with DbleXtendedPower thanks to a well-known and efficient test [Ziv 1991; Daramy-Loirat et al. 2006].

3.6 When n Is a Constant

Frequently n is a constant, that is, n is known at compile-time. In such a case, it is possible to simplify the iterated product algorithm, as well as the two-step algorithm (that uses algorithm DbleXtendedPower first, and the other algorithm only if the extended-precision result does not make it possible to deduce a correctly rounded value). The possible simplifications are as follows:

—the loops can be unrolled, there is no longer any need to perform the computations "$i := \lfloor i/2 \rfloor$", nor to do tests on variable i;

—moreover, for the first values of n, addition chains to obtain the minimal number of multiplications needed to compute a power are known. This can be used for optimizing the algorithm. For instance, for n up to 10001, such addition chains can be obtained online.[5]

Since all the branches are resolved a priori, some obvious optimizations can also be performed without any (possibly costly) additional branch in the code. For instance, at the first iteration of the loop in Algorithm 5, since we know that $v = 0$, the statement $[u, v] := \mathit{DblMult}(u, v, u, v)$ is replaced by $[u, v] := \mathit{FastMult}(u, u)$.

4. AN ALGORITHM BASED ON LOGARITHMS AND EXPONENTIALS

With binary asymptotic complexity in mind [Brent 1976], it might seem silly to compute x^n by

$$x^n = 2^{n \cdot \log_2 x}.$$

However, in this section, we are going to show that, on actual superscalar and pipelined hardware, if n is large enough, the situation is different. For that purpose, we consider an implementation on the Itanium architecture. Itanium offers both extended precision and the fma instruction, as well as some other useful operations. These features permit achieving high performance: in the timings reported in Section 5, the measured average evaluation time for x^n is equivalent to about 21 sequential multiplications on Itanium 2.

4.1 Basic Layout of the Algorithm

We combine the scheme for x^n based on logarithms and exponentials with a two-step approximation approach. This approach has already been proven efficient for common correctly rounded elementary functions [Gal 1986; Ziv 1991;

[5]http://www.research.att.com/ njas/sequences/b003313.txt.

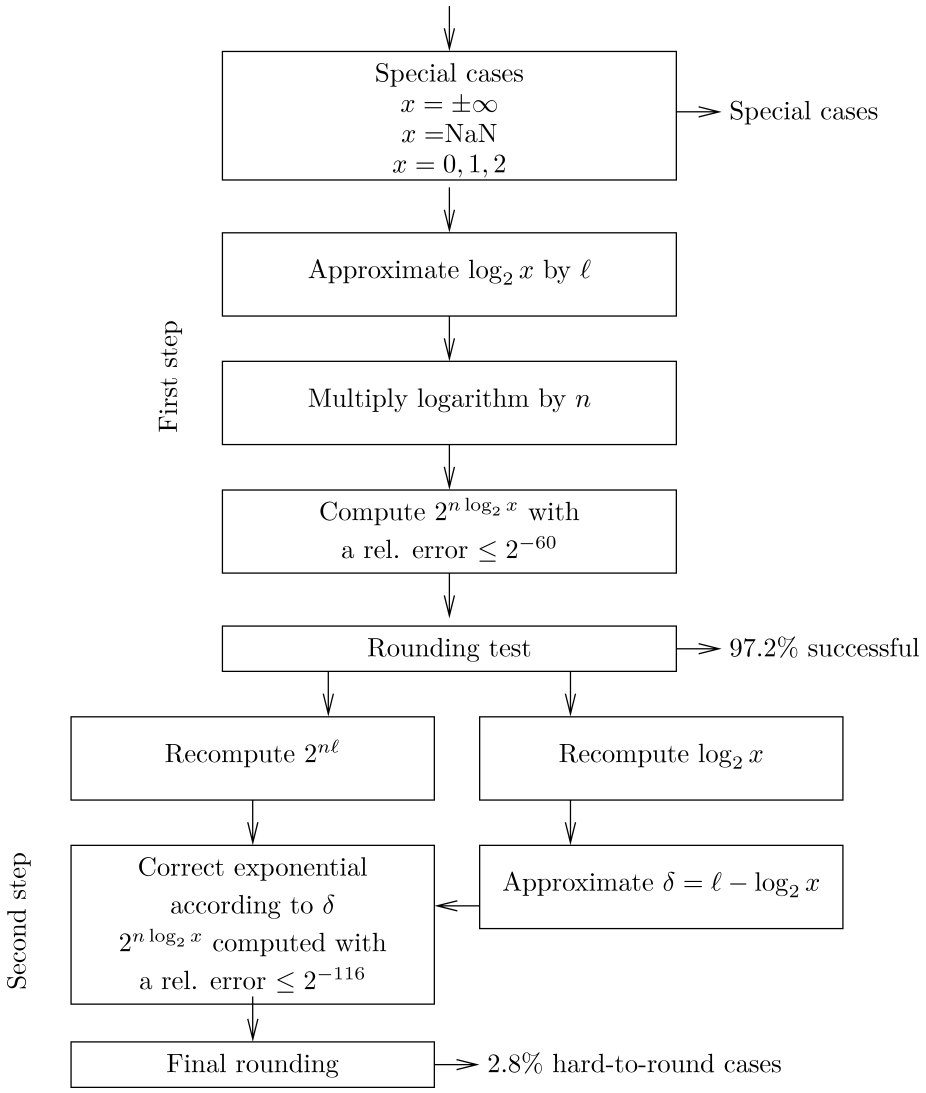

Fig. 2. Two-step exponential of logarithm approach.

de Dinechin et al. 2005]. It is motivated by the rarity of hard-to-round cases. In most cases, an approximation which is just slightly more accurate than the final precision suffices to ensure correct rounding. Only in rare cases, the result of the function must be approximated up to the accuracy demanded by the worst cases [Gal 1986; Ziv 1991; de Dinechin et al. 2005]. There is a well-known and efficient test whether correct rounding is already possible with small accuracy [Ziv 1991; Daramy-Loirat et al. 2006]. We prove in the sequel that our scheme computes x^n correctly rounded for n up to 733.

The proposed scheme is shown in Figure 2. The function $2^{n \cdot \log_2(x)}$ is first approximated with an accuracy of $2^{-59.17}$. These 6.17 guard bits with respect

to double precision make the hard-to-round-case probability as small as about $2 \cdot 2^{-6.17} \approx 2.8\%$, which means that the first step is successful with probability 97.2%. If rounding is not possible, correct rounding is ensured by the second step, which provides an accuracy of 2^{-116}.

We have designed the second step to take advantage of the superscalar capabilities of the Itanium processor. Indeed, as the approximate logarithm $\ell = \log_2(x) + \delta$ computed at the first step is already available, it is possible to perform in parallel the computation of the more accurate logarithm and exponential. For this purpose, we write

$$x^n = 2^{n \log_2(x)} = 2^{n\ell} 2^{-n\delta},$$

where δ denotes as before the error in the approximation ℓ to $\log_2(x)$ computed at the first step. In the second step, a new approximation to $2^{n\ell}$ is computed with a relative error bounded by 2^{-117}. In parallel, $\log_2(x)$ is computed with a corresponding accuracy, and an approximate δ is deduced. The multiplication of $2^{n\ell}$ by $2^{-n\delta}$ is then approximated using the first-order expansion $2^{-n\delta} = 1 - cn\delta$. After this final correction step, the relative error in the computed value of x^n is bounded by 2^{-116}, which is sufficient to ensure correct rounding for n up to 773 according to the worst cases reported in Table I.

The logarithm and the exponential approximation subalgorithms follow the well-known principles of table lookup and polynomial approximation, both in the first and the second step. The algorithms implemented are variants of the techniques presented in [Wong and Goto 1994; Cornea et al. 2002; Lauter 2003; de Dinechin et al. 2007]. Our implementation uses about 8 kbytes of tables. The approximation polynomials have optimized floating-point coefficients [Brisebarre and Chevillard 2007].

In the sequel of the section, we give details on the implementation of the logarithm and the exponential, and a sketch of the error analysis to bound the error in the computed value of x^n.

4.2 Implementation of the Logarithm

Both in the first and in the second step, the logarithm $\log_2 x$ is based on the following argument reduction:

$$\begin{aligned}
\log_2 x &= \log_2\left(2^E \cdot m\right) \\
&= E + \log_2(m \cdot r) - \log_2 r \\
&= E + \log_2\left(1 + \left(m \cdot r - 1\right)\right) - \log_2 r \\
&= E + \log_2\left(1 + z\right) + \log_2 r \\
&= E + p(z) + logtblr[m] + \delta.
\end{aligned}$$

In this argument reduction, the decomposition of x into E and m can be performed using Itanium's `getf` and `fmerge` instructions [Cornea et al. 2002].

The value r is produced by Itanium's `frcpa` instruction. This instruction gives an approximate to the reciprocal of m with at least 8.886 valid bits [Cornea et al. 2002]. The instruction is based on a small table indexed by the first 8 bits of the significand (excluding the leading 1) of x. This makes it possible to tabulate

the values of $\log_2 r$ in a table indexed by these first 8 bits of the significand of m.

The reduced argument z can exactly be computed with an fma:

$$z = RN_e\left(m \cdot r - 1\right).$$

Indeed, as can easily be verified on the 256 possible cases, the `frcpa` instruction [Cornea et al. 2002] returns its result r on floating-point numbers with at most 11 leading nonzero significand bits. Since x is a double-precision number, $x \cdot r$ holds on $53 + 11 = 64$ bits, and hence is an extended-precision number. No rounding occurs on the subtraction $x \cdot r - 1$ as per Sterbenz' lemma [Sterbenz 1974].

The exactness of the reduced argument z makes it possible to reuse it in the second, more accurate step of the algorithm. It is worth to remark that the latency for obtaining the reduced argument is small. It is produced by only three depending operations: `fmerge`, `frcpa`, and `fms`.

The tabulated values $logtbl[m]$ for $\log_2 r$ are stored as a sum of two extended-precision numbers: $logtblr_{hi}[m] + logtblr_{lo}[m]$. The absolute error of the entries with respect to the exact value $\log_2 r$ is bounded by 2^{-130}. Both extended-precision numbers of an entry are read in the first step of the algorithm. The second step can reuse the values directly.

The magnitude of the reduced argument z is bounded as follows: we have $r = \frac{1}{m} \cdot \left(1 + \varepsilon_r\right)$ with $|\varepsilon_r| \leq 2^{-8.886}$. Hence

$$z = m \cdot r - 1 = \frac{1}{m} \cdot \left(1 + \varepsilon_r\right) \cdot m - 1 = \varepsilon_r$$

is bounded by $2^{-8.886}$.

The function $\log_2\left(1 + z\right)$ is approximated using a polynomial of degree 6 for the first step and of degree 12 for the second step. The corresponding absolute approximation errors are bounded by $2^{-69.49}$ respectively $2^{-129.5}$. The polynomials have optimized floating-point coefficients. We minimize the number of extended-precision and double-precision coefficients as follows: double-precision numbers are preferred to extended-precision numbers and single-precision numbers are preferred to double-precision numbers. The reason is that memory load times on Itanium increase with increasing precision of the constants loaded. For instance, double-precision numbers can be loaded twice as fast as extended-precision numbers [Cornea et al. 2002; Markstein 2000].

The approximation polynomials are evaluated using a mixture of Estrin and Horner scheme [Cornea et al. 2002; Muller 2006; de Dinechin et al. 2007; Revy 2006]. In the first step, extended precision is sufficient for obtaining an absolute roundoff error less than 2^{-70}. In the second, accurate step, extended-extended arithmetic is used: each number is represented as a pair of extended-precision floating-point numbers, and variants of the DblMult algorithm are used for the aritmetic operations. This yields an absolute roundoff error less than 2^{-130}.

Reconstruction of the value of $\log_2$ out of the polynomial approximation and the table values is performed in extended-extended arithmetic in both steps. The value of $\log_2 x$ is returned in three registers E, ℓ_{hi} and ℓ_{lo}. In the first step,

a modified version of the Fast2Sum algorithm is used that ensures that ℓ_{hi} is written only on 53 bits (double precision).

4.3 Implementation of the Exponential

The following argument reduction is used for the exponential function 2^t:

$$\begin{aligned} 2^t &= 2^M \cdot 2^{t-M} \\ &= 2^M \cdot 2^{i_1 \cdot 2^{-7}} \cdot 2^{i_2 \cdot 2^{-14}} \cdot 2^{t-(M+i_1 \cdot 2^{-7}+i_2 \cdot 2^{-14})} \\ &= 2^M \cdot 2^{i_1 \cdot 2^{-7}} \cdot 2^{i_2 \cdot 2^{-14}} \cdot 2^{u-\Delta} \\ &= 2^M \cdot exptbl_1[i_1] \cdot \left(1 + exptbl_2[i_2]\right) \cdot q(u) \cdot \left(1+\varepsilon\right). \end{aligned}$$

Here the values M, i_1 and i_2 are integers. They are computed from

$$t = n \cdot E + n \cdot \ell_{hi} + n \cdot \ell_{lo}$$

using the fma, shifts, and Itanium's `getf` instruction giving the significand of a number as follows:

$$\begin{aligned} s &= RN_e\left(n \cdot \ell_{hi} + \left(2^{49}+2^{48}\right)\right), \\ a &= RN_e\left(s - \left(2^{49}+2^{48}\right)\right) = \left\lfloor n \cdot \ell_{hi} \cdot 2^{14} \right\rceil, \\ b &= RN_e\left(n \cdot \ell_{hi} - a\right), \\ u &= RN_e\left(n \cdot \ell_{lo} + b\right), \\ k &= \texttt{getf}(s), \\ M &= k \div 2^{14}, \\ i_1 &= \left(k \div 2^7\right) \bmod 2^7, \\ i_2 &= k \bmod 2^7. \end{aligned}$$

In this sequence, all floating-point operations except those producing s and u are exact by Sterbenz' lemma [Sterbenz 1974]. The error in s is compensated in the following operations; actually, it is b. The absolute error Δ the value u is affected of is bounded by 2^{-78} because u is upper-bounded by 2^{-15}.

For approximating 2^u for $u \in \left[-2^{-15}; 2^{-15}\right]$, a polynomial of degree 3 is used in the first step and a polynomial of degree 6 is the second, accurate step. The polynomials provide a relative accuracy of $2^{-62.08}$ respectively $2^{-118.5}$.

The table values $exptbl_1[i_1]$ and $exptbl_2[i_2]$ are all stored as extended-extended numbers. Only the higher parts are read in the first step. The second step reuses these higher parts and reads the lower parts. The reconstruction is performed with extended-precision multiplications in the first step and with DblMult in the second step.

The first step delivers the final result $2^{n \cdot \log_2 x} \cdot \left(1+\varepsilon_1\right)$ as two floating-point numbers r_{hi} and r_{lo}. The value r_{hi} is a double-precision number; hence r_{lo} is a roundoff error estimate of rounding x^n to double precision.

In the second step, the exponential $2^{n \cdot \ell}$ is corrected by

$$2^{\delta''} = 2^{n \cdot (E+i_1 \cdot 2^7+i_2 \cdot 2^{14}+u)-n \cdot (E+\ell')} = 2^{\delta-\Delta},$$

where ℓ' is an accurate approximation to the logarithm. The correction first approximates $\delta'' = n \cdot \left(E + i_1 \cdot 2^7 + i_2 \cdot 2^{14} + u\right) - n \cdot \left(E + \ell'\right)$ up to 58 bits and then uses a polynomial of degree 1 for approximating the correction $2^{\delta''}$. The final result is delivered as an extended-extended value.

The function x^n has some arguments for which it is equal to the midpoint of two consecutive double-precision numbers, an example is 9^{17}. For rounding correctly in that case, the tie-to-even rule must be followed. The final rounding after the second accurate approximation step must hence distinguish the two cases. The separation is easy because the worst cases of the function are known: if and only if the approximation is nearer to a midpoint than the worst case of the function, the infinitely exact value of the function is a midpoint. See Lauter and Lefvre [2009] for details on the technique.

4.4 Complete Error Bounds

A complete, perhaps formal proof of the error bounds for the two steps of the algorithm goes beyond the scope of this article. Following the approach presented in de Dinechin, Lauter, and Melquiond [2006] and Daramy-Loirat, Defour, de Dinechin, Gallet, Gast, Lauter, and Muller [2006], the Gappa tool can be used for this task. Bounds on approximation errors can safely be certified using approaches found in Chevillard and Lauter [2007]. Let us give just the general scheme of the error bound computation and proof for the first step of the presented algorithm.

We are going to use the following notations:

—$E + \ell = E + \ell_{hi} + \ell_{lo}$ stands for the approximation to the logarithm, $\log_2 x$,
—δ is the associated total absolute error;
—δ_{table}, δ_{approx}, δ_{eval}, and $\delta_{reconstr}$ are the absolute errors of the tables, the approximation, the evaluation and reconstruction of the logarithm;
—$r_{hi} + r_{lo}$ stands for the approximation to $x^n = 2^{n \cdot \log_2 x}$;
—$\varepsilon_{firststep}$ is the associated total relative error;
—ε_1 is the total relative error due only to the approximation to the exponential without the error of the logarithm;
—ε_{table}, ε_{approx}, ε_{eval}, and $\varepsilon_{reconstr}$ are the relative errors of the tables, the approximation, the evaluation, and reconstruction of the exponential, and
—Δ stands for the absolute error the reduced argument u of the exponential is affected with.

The following error bound can hence be given:

$$\begin{aligned} r_{hi} + r_{lo} &= 2^M \cdot 2^{i_1 \cdot 2^{-7}} \cdot 2^{i_2 \cdot 2^{-14}} \cdot p(u) \cdot \\ &\quad \cdot (1 + \varepsilon_{reconstr}) \cdot (1 + \varepsilon_{table}) \cdot (1 + \varepsilon_{eval}) \\ &= 2^M \cdot 2^{i_1 \cdot 2^{-7}} \cdot 2^{i_2 \cdot 2^{-14}} \cdot 2^{u - \Delta} \\ &\quad \cdot 2^{\Delta} \cdot (1 + \varepsilon_{reconstr}) \cdot (1 + \varepsilon_{table}) \cdot (1 + \varepsilon_{eval}) \cdot (1 + \varepsilon_{approx}) \\ &= 2^{n \cdot (E + \ell_{hi} + \ell_{lo})} \cdot (1 + \varepsilon_1), \end{aligned}$$

where ε_1 is bounded by

$$|\varepsilon_1| \leq \varepsilon_{reconstr} + \varepsilon_{table} + \varepsilon_{eval} + \varepsilon_{approx} + 2 \cdot \Delta + \mathcal{O}(\varepsilon^2).$$

With $|\varepsilon_{reconstr}| \leq 3 \cdot 2^{-64}$, $|\varepsilon_{table}| \leq 3 \cdot 2^{-64}$, $|\varepsilon_{eval}| \leq 4 \cdot 2^{-64}$, $\left|\varepsilon_{approx}\right| \leq 2^{-62.08}$, and $|\Delta| \leq 2^{-78}$, this gives

$$|\varepsilon_1| \leq 2^{-60.5}.$$

Additionally, we obtain for $E + \ell_{hi} + \ell_{lo}$:

$$\begin{aligned} E + \ell_{hi} + \ell_{lo} &= E + logtblr_{hi}[m] + logtblr_{lo}[m] + p(z) + \delta_{eval} + \delta_{reconstr} \\ &= E + \log_2(r) + \log_2(1+z) + \delta_{table} + \delta_{approx} + \delta_{eval} + \delta_{reconstr} \\ &= \log_2(x) + \delta_{table} + \delta_{approx} + \delta_{eval} + \delta_{reconstr} \\ &= \log_2(x) + \delta. \end{aligned}$$

Since $\left|\delta_{approx}\right| \leq 2^{-69.49}$, $|\delta_{eval}| \leq -\log_2\left(1 - 2^{-8.886}\right) \cdot 3 \cdot 2^{-64} \leq 2^{-70.7}$, $|\delta_{table}| \leq 2^{-128}$, and $|\delta_{reconstr}| \leq 2^{-117}$, we get

$$|\delta| \leq 2^{-68.9}.$$

These bounds eventually yields

$$\begin{aligned} r_{hi} + r_{lo} &= 2^{n \cdot (\ell_{hi} + \ell_{lo})} \cdot (1 + \varepsilon_1) \\ &= 2^{n \cdot \log_2(x)} \cdot 2^{n \cdot \delta} \cdot (1 + \varepsilon_1) \\ &= x^n \cdot (1 + \varepsilon_{firststep}). \end{aligned}$$

With $n \leq 733$, this gives

$$\left|\varepsilon_{firststep}\right| \leq 2^{733 \cdot 2^{-68.9}} \cdot \left(1 + 2^{-60.5}\right) - 1 \leq 2^{-59.17}.$$

For the second step, a similar error bound computation can be performed. One deduces that the overall relative error ε_2 of the second step is bounded by $|\varepsilon_2| \leq 2^{-116}$. This is sufficient for guaranteeing correct rounding for n up to 733.

5. COMPARISONS AND TESTS

In this section, we report timings for the various algorithms described above. The algorithms have been implemented on the Intel/HP Itanium architecture, using the Intel ICC compiler.[6] We compared the following programs:

—IteratedProductPower: our implementation follows Algorithm 5 strictly.

—Two-step IteratedProductPower: for the first, "fast" step we use Algorithm 4, and if needed, for the "accurate" step we use Algorithm 5 (see Section 3.5).

—IteratedProductPower and two-step IteratedProductPower with constant n: these implementations are the same as the two previous ones, except that the exponent n is fixed a priori (see Subsection 3.6). To take this information into account, a different function has been implemented for each exponent n considered. In the experiments reported hereafter, we consider exponents n

[6]We used ICC v10.1, on an Itanium 2-based computer.

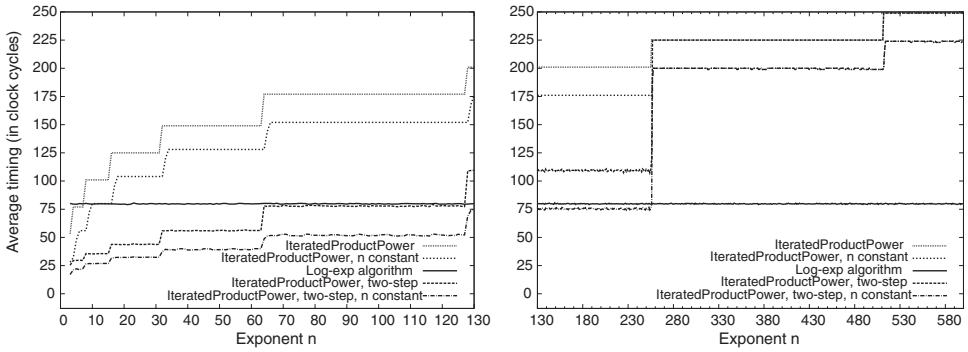

Fig. 3. Average-case timings.

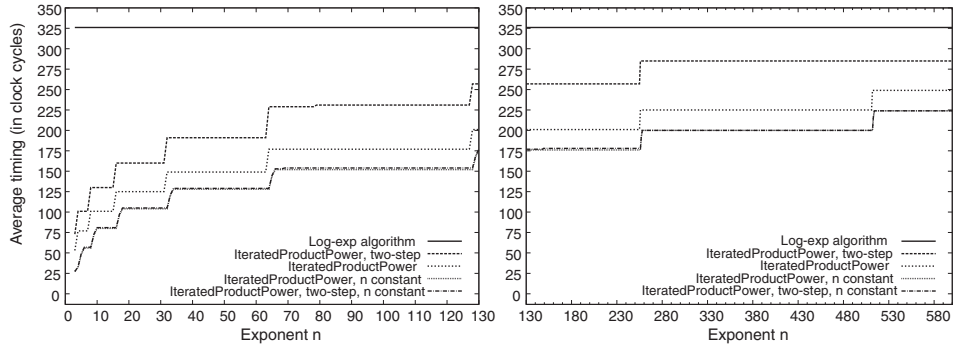

Fig. 4. Worst-case timings.

ranging from 3 to 600: we have used an automated code generator to produce C code for each of these functions.

—Log-exp algorithm described in Section 4: let us underline that this implementation, in contrast to the others, checks for special cases in input, such as $\pm\infty$ or NaN; subnormal rounding is also supported. This slightly favors the other implementations in the timings.

First we consider the average timings for computing x^n rounded to the nearest floating-point value. For each n from 3 to 600, we compute the average execution time for 16384 double-precision arguments randomly generated in the interval $[1, 2]$. We report in Figure 3 the average timings over these arguments with respect to n.

We also report in Figure 4 the worst-case timings. For a function based on the two-step approach, the worst-case timing is the timing of a call to this function with an input x that causes the second step to be executed (the input x need not be a worst case as defined in Section 2 for this). We also recall in Figure 4 the timings for the other functions for comparison purposes.

Table V. Timings (in Clock Cycles) for Tested Functions

n	Iterated product power	Two-step iterated		Log-exp		Iterated (n fixed)	Two-step iterated (n fixed)	
		Average	Worst	Average	Worst		Average	Worst
3	53	25.3	73	80.1	326	28	17.1	27
4	77	29.5	101	79.5	326	32	21.1	33
5	77	29.6	101	79.4	326	48	22.2	49
6	77	29.7	101	80.0	326	56	21.3	57
7	77	29.6	101	79.7	326	56	22.3	57
8	101	35.7	130	80.2	326	56	26.5	57
9	101	35.5	130	79.7	326	72	26.8	73
10	101	35.4	130	79.6	326	80	26.8	81
15	101	35.5	130	79.7	326	80	26.9	81
16	125	43.8	160	79.9	326	80	31.6	81
17	125	43.8	160	79.6	326	96	32.1	97
18	125	43.7	160	79.5	326	104	32.3	105
31	125	44.0	160	80.0	326	104	32.5	105
32	149	55.8	191	79.7	326	104	37.4	105
33	149	55.9	191	80.0	326	120	39.4	121
34	149	56.0	191	79.4	326	128	39.0	129
63	149	56.1	191	80.0	326	128	40.0	129
64	177	78.0	229	80.0	326	128	48.7	129
65	177	77.4	229	80.2	326	144	51.2	145
66	177	77.9	229	80.3	326	152	51.7	153
127	177	77.3	231	79.4	326	152	52.2	154
128	201	109.1	257	79.6	326	152	68.9	154
129	201	109.3	257	79.8	326	168	73.8	169
130	201	109.6	257	79.8	326	176	75.2	177
255	201	109.8	257	79.6	326	176	76.1	178
256	225	119.5	285	79.4	326	176	78.2	178
257	225	225.0	285	79.8	326	192	192.0	192
258	225	225.0	285	80.1	326	200	200.0	200
511	225	225.0	285	79.4	326	200	199.0	200
512	249	249.0	285	79.8	326	200	200.0	200
513	249	249.0	285	79.9	326	216	216.0	216
514	249	249.0	285	80.3	326	224	224.0	224
600	249	249.0	285	79.6	326	224	224.0	224

Finally, the timings for some typical values of the exponent n are reported in Table V: the timings for each "step" observed in the graphics of Figures 3 and 4 can be read in this table.

From these timings, we can see that, in the worst cases, the implementations based on iterated products are always more efficient than the one based on the log-exp. Moreover, if we consider the average timings reported in Figure 3 and in Table V, we can make the following observations:

—The average, and worst-case timings for the log-exp implementation are constant with respect to n, with an average execution time of about 80 clock cycles and a worst case execution time of 236 cycles.

—The straightforward (one-step) IteratedProductPower algorithm is more efficient on average than the log-exp only for $n \leq 9$.
—The implementations of the two-step IteratedProductPower algorithm with n fixed are significantly more efficient than the log-exp approach as long as $n \leq 128$.

6. CONCLUSIONS

We have introduced several algorithms for computing x^n, where x is a double-precision floating-point number, and n is an integer, $3 \leq n \leq 733$. Our multiplication-based algorithms require the availability of a fused multiply-add (fma) instruction, and an extended-precision format for intermediate calculations.

According to our experiments, the best choice depends on the order of magnitude of the exponent n, on whether n is known at compile-time or not, and on whether one is interested in the best:

—Worst-case performance. Then Algorithm 5 (IteratedProductPower) is preferable (at least, up to $n = 733$, since we do not have a proof that our algorithms work for larger values);
—Average-case performance and n is *not* a constant. Then the two-step IteratedProductPower algorithm should be used for n less than around 60, and the log-exp algorithm should be used for larger values of n;
—Average-case performance and n is a constant. Then the "specialized" two-step IteratedProductPower algorithm should be used for n less than around 250, and the log-exp algorithm should be used for larger values of n.

REFERENCES

American National Standards Institute and Institute of Electrical and Electronic Engineers. 1985. *IEEE Standard for Binary Floating-Point Arithmetic*, (ANSI/IEEE Standard 754-1985). ANSI/IEEE, New York, NY.

American National Standards Institute and Institute of Electrical and Electronic Engineers. 1987. *IEEE Standard for Radix Independent Floating-Point Arithmetic*, (ANSI/IEEE Standard 854-1987). ANSI/IEEE, New York, NY.

Brent, R. P. 1976. Fast multiple-precision evaluation of elementary functions. *J. ACM 23,* 2, 242–251.

Brisebarre, N. and Chevillard, S. 2007. Efficient polynomial L^∞ approximations. In *Proceedings of the 18th IEEE Symposium on Computer Arithmetic* (ARITH18). IEEE Computer Society Press, Los Alamitos, CA, 169–176.

Chevillard, S. and Lauter, C. 2007. A certified infinite norm for the implementation of elementary functions. In *Proceedings of the 7th International Conference on Quality Software*. IEEE Computer Society Press, Los Alamitos, CA, 153–160.

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

Daramy-Loirat, C., Defour, D., de Dinechin, F., Gallet, M., Gast, N., Lauter, C. Q., and Muller, J.-M. 2006. Cr-libm, a library of correctly-rounded elementary functions in double-precision. Tech. rep. Arenaire team, LIP Laboratory, ENS Lyon, Lyon, France. `https://lipforge.ens-lyon.fr/frs/download.php/99/crlibm-0.18beta1.pdf`. Dec.

de Dinechin, F., Ershov, A., and Gast, N. 2005. Towards the post-ultimate libm. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic* (ARITH17). IEEE Computer Society Press, Los Alamitos, CA.

DE DINECHIN, F., LAUTER, C. Q., AND MELQUIOND, G. 2006. Assisted verification of elementary functions using Gappa. In *Proceedings of the 21st Annual ACM Symposium on Applied Computing—MCMS Track*, P. Langlois and S. Rump, Eds. Vol. 2. (ACM) Press, New York, NY, 1318–1322.

DE DINECHIN, F., LAUTER, C. Q., AND MULLER, J.-M. 2007. Fast and correctly rounded logarithms in double-precision. *RAIRO, Theoret. Informat. and Appl. 41*, 85–102.

DEKKER, T. J. 1971. A floating-point technique for extending the available precision. *Numer. Math. 18*, 224–242.

GAL, S. 1986. Computing elementary functions: A new approach for achieving high accuracy and good performance. In *Accurate Scientific Computations*. Lecture Notes in Computer Science, vol. 235. Springer, Berlin, Germany, 1–16.

IEEE. 2008. *IEEE Standard for Floating-Point Arithmetic*. (IEEE Standard 754^{TM}-2008). IEEE, New York, NY.

KNUTH, D. 1998. *The Art of Computer Programming*, 3rd ed. Vol. 2. Addison-Wesley, Reading, MA.

LAUTER, C. 2003. A correctly rounded implementation of the exponential function on the Intel Itanium architecture. Tech. Rep. RR-5024, INRIA. Nov. http://www.inria.fr/rrrt/rr-5024.html.

LAUTER, C. AND LEFVRE, V. 2009. An efficient rounding boundary test for `pow(x,y)` in double precision. *IEEE Trans. on Comput. 58,* 2 (Feb.), 197–207.

LEFÈVRE, V. 1999. *Developments in Reliable Computing*. Kluwer Academic Publishers, Dordrecht, The Netherlands. Chapter "An Algorithm That Computes a Lower Bound on the Distance Between a Segment and $\mathbb{Z}^2$," 203–212.

LEFÈVRE, V. 2000. Moyens arithmétiques pour un calcul fiable. Ph.D. dessertation. École Normale Supérieure de Lyon, Lyon, France.

LEFÈVRE, V. 2005. New results on the distance between a segment and $\mathbb{Z}^2$. Application to the exact rounding. In *Proceedings of the 17th IEEE Symposium on Computer Arithmetic* (ARITH17). IEEE Computer Society Press, Los Alamitos, CA, 68–75.

LEFÈVRE, V. AND MULLER, J.-M. 2001. Worst cases for correct rounding of the elementary functions in double precision. In *Proceedings of the 15th IEEE Symposium on Computer Arithmetic* (ARITH15). IEEE Computer Society Press, Los Alamitos, CA.

LI, R.-C., MARKSTEIN, P., OKADA, J., AND THOMAS, J. 2002. The libm library and floating-point arithmetic in hp-ux for itanium 2. Tech. rep. Hewlett-Packard, Palo Alto, CA. `http://h21007.www2.hp.com/dspp/files/unprotected/libm.pdf`.

MARKSTEIN, P. 2000. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, Englewood Cliffs, NJ.

MØLLER, O. 1965. Quasi double-precision in floating-point addition. *BIT 5*, 37–50.

MULLER, J.-M. 2006. *Elementary Functions, Algorithms and Implementation*, 2nd ed. Birkhäuser Boston, MA.

OGITA, T., RUMP, S. M., AND OISHI, S. 2005. Accurate sum and dot product. *SIAM J. Sci. Comput. 26,* 6, 1955–1988.

REVY, G. 2006. Analyse et implantation d'algorithmes rapides pour l'évaluation polynomiale sur les nombres flottants. M.S. dessertation. École Normale Supérieure de Lyon, Lyon, France.

RUMP, S. M., OGITA, T., AND OISHI, S. 2005–2008. Accurate floating-point summation part I: Faithful rounding. *SIAM J. Sci. Comput.*. To appear.

STERBENZ, P. H. 1974. *Floating Point Computation*. Prentice-Hall, Englewood Cliffs, NJ.

WONG, W. F. AND GOTO, E. 1994. Fast hardware-based algorithms for elementary function computations using rectangular multipliers. *IEEE Trans. Comput. 43,* 3 (Mar.), 278–294.

ZIV, A. 1991. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Trans. Math. Softw. 17,* 3 (Sep.), 410–423.

Received May 2008; revised January 2009; accepted May 2009