Exact and mid-point rounding cases of power(x,y)

Christoph Lauter

Laboratoire de l'Informatique du Parallélisme

École Normale Supérieure de Lyon
Unité Mixte de Recherche CNRS-INRIA-ENS LYON-UCBL n° 5668

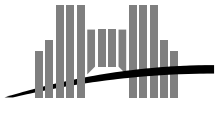
Exact and mid-point rounding cases of `power(x,y)`

Lauter, Christoph Quirin

November 2006

Research Report N° RR2006-46

École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : +33(0)4.72.72.80.37
Télécopieur : +33(0)4.72.72.80.80
Adresse électronique : `lip@ens-lyon.fr`



Exact and mid-point rounding cases of `power(x,y)`

Lauter, Christoph Quirin

November 2006

Abstract

Correct rounding of the `power` function is currently based on an iterative process computing more and more accurate intermediate approximations to x^y until rounding correctly becomes possible. It terminates iff the value of the function is not exactly a floating-point number or an midpoint of two floating-point numbers of the format. For other elementary functions such as e^x, arguments for which $f(x)$ is such an exact case are, few. They can be filtered out by simple tests. The power function has at least 2^{35} such arguments. Simple tests on x and y do not suffice here.

This article presents an algorithm for performing such an exact case test. It is combined with an approach that allows for fast rejection of cases that are not exact or mid-point. The correctness is completely proven. It makes no usage of costly operations such as divisions, remainders or square roots as previous approaches do.

The algorithm yields a speed-up of 1.8 on average in comparison to another implementation for the same final target format. It reduces also the percentage of average time needed for the exactness test from 38% at each call to 31% under unlikely conditions.

The algorithm is given for double precision but adapts and scales perfectly for higher precisions such as double-extended and quad precision. Its complexity is linear in the precision.

Keywords: Correct rounding, elementary functions, exact rounding cases, power function

Résumé

L'arrondi correct de la fonction `power` est actuellement basé sur un processus itératif qui calcule une approximation intermédiaire à x^y de plus en plus précise et qui peut s'arrêter dès que l'arrondi correct devient possible. Ce processus termine ssi la valeur de la fonction n'est pas exactment un nombre flottant ou le milieu de deux nombres flottants successifs. Pour d'autres fonctions élémentaires telles que e^x, le nombre d'arguments pour lesquels $f(x)$ est un tel cas exact, est petit. Ils peuvent donc être filtrés par de simples tests. La fonction power a au moins 2^{35} de tels cas. Des tests simples sur x et y n'y suffisent plus.

Cet article présente un algorithme pour un tel test de cas exacts. Une approche qui permet un rejet rapide de cas qui ne sont pas exacts ou mi-ulps s'ajoute à l'algorithme donné. Sa correction est complètement prouvée. Il n'utilise pas d'opérations coûteuses telles que divisions, restes ou racines carrées comme employées par les approches précédentes.

L'algorithme permet d'obtenir une accéleration du calcul en moyenne d'un facteur 1.8 par rapport à une autre implémentation pour le même format final. Son utilisation réduit aussi le pourcentage de temps moyen nécessaire pour le test des cas exacts de 38% pour chaque appel à 31% sous des conditions improbables.

L'algorithme est donné pour la précision double mais il peut être adapté parfaitement à des précisions plus grandes comme les précisions double-étendue ou quad, sans qu'il y ait un problème de passage à l'échelle. Sa complexité est linéaire dans la précision.

Mots-clés: Arrondi correct, fonctions élémentaires, cas d'arrondi exacts, fonction power

1 Introduction

Correct rounding of a function means providing, given a rounding mode $\circ(x)$, for each argument x its rounded value $\circ(f(x))$ exactly as if the function $f(x)$ were computed in infinite precision and then rounded. If some of the values of the function are not rational, the explicite computation of an infinitely precise intermediate value is clearly impossible. Nevertheless, the correct rounding of elementary, univariate, transcendental functions, such as e.g. e^x or $\sin(x)$, has been shown to be feasible and performant for at least double precision in the last years [15, 6, 1, 4]. Previous state of the art allowed only the correct rounding of 5 basic operations, addition, substraction, multiplication, division and square root extraction, $\oplus$, $\ominus$, $\otimes$, $\oslash$, `sqrt`.

In fact, the original version IEEE 754 standard [2] for binary floating-point arithmetic requires only these last five operations to be correctly rounded in the four rounding modes the standard defines. These modes are round-to-nearest-ties-to-even, round-up, round-down and round-towards-zero. In contrast, the original standard does not tell anything on elementary functions. However, the IEEE 754 standard is currently undergoing a revision process[1]. The current draft of the revised standard comprises an annex requiring some transcendental functions to be correctly rounded.

The main problem that has longtime prevented the standardization of correctly rounded elementary functions is the so-called Table Marker's Dilemma [12, 11]. In finite time the infinitely exact value $\hat{y} = f(x)$ of a transcendental function $f(x)$ can only be approximated. Correct rounding $\circ(f(x))$ of the function is possible only if it can be shown that for some approximation $y = \hat{y} \cdot (1 + \varepsilon)$, the rounding of the approximation, $\circ(y)$ is equal to the rounding of the exact value. Formally it means $\circ(f(x) \cdot (1 + \varepsilon)) = \circ(f(x))$. The roundings may be different if the the approximation interval $[\hat{y} \cdot (1 - \varepsilon); \hat{y} \cdot (1 + \varepsilon)]$ induced by ε contains a so-called rounding boundary, i.e. a discontinuity of the rounding function $\circ(x)$. For round-to-nearest mode, rounding boundaries are the exact midpoints between floating-point numbers of the format to be rounded to. It can be shown that for elementary functions $f(x)$ there exists always some $\overline{\varepsilon}$ such that $|\varepsilon| \leq \overline{\varepsilon}$ implies correct rounding of an approximation with relative error ε. The actual value of $\overline{\varepsilon}$ is a priori unknown and difficult to compute. The dilemma is that the accuracy ε of the intermediate result before rounding may be much higher than the final accuracy of the result after rounding. The rare theoretical estimates of $\overline{\varepsilon}$ are extremely pessimistic [13].

Nevertheless, for a particular given input x, the minimal value of ε necessary to guarantee the correct rounding of $f(x)$ may not be too small. An approach for overcoming the Table Maker's Dilemma is Ziv's onion peeling strategy [12, 15]. In an iterative process the given function is approximated more and more precisely, i.e. ε is decreased, until the rounding becomes possible because no rounding boundary lies in the approximation interval. If it can be shown that for a given input x, $f(x)$ is not exactly equal to a rounding boundary, there exists always a value $\varepsilon \neq 0$ such that the process terminates.

Such an iterative approach provides good average performance results. In binary arithmetic, the fact that $f(x)$ is near to a rounding boundary implies that there is a long series of zeros or ones in the infinite precision significant of $f(x)$ [12]. The bits after some rank in this significant can be considered to be randomly distributed [8]. Hard cases of the Table Maker's Dilemma with very small ε are thus extremely rare. Hence average performance is pratically

[1] `http://grouper.ieee.org/groups/754/`

not affected by the consecutive iterations of the process because the first step decides the rounding in the very most cases.

Means are known that allow computing $\overline{\varepsilon}$ for simple, univariate, elementary functions for a rounding to double precision [14]. In this case the iterative process can be statically limited to some number of iterations, typically 2. The result is then known to be roundable at least after the second step. This concept is e.g. at the base of the `crlibm` library for correct rounding elementary functions in double precision [1]. If $\overline{\varepsilon}$ is known, filtering out arguments with images lying on rounding boundaries may be not necessary. The function $\log_{10}(x)$ in double precision is an example [1].

In other cases, in order to guarantee the termination of Ziv's iterative process, filtering exact and mid-point cases is necessary. It is very simple for most elementary, transcendental functions, such as e^x, $\sin(x)$, 2^x etc. Since floating-point numbers are algebraic, the images of almost all inputs are transcendental due to Lindemann's theorem [9]. These transcendental images can thus not be rounding boundaries, which are floating-point numbers in a 1 bit more precise format. For example, $e^x \in \mathbb{F}_p$ only for $x = 0$ or $2^x \in \mathbb{F}_p$ only if $x \in \mathbb{Z}$.

The `power` function, defined as `power`: $\mathbb{F}_p \times \mathbb{F}_p \to \mathbb{F}_p$, $(x, y) \mapsto \circ_p(x^y)$, is different from usual elementary, transcendental functions in many ways. Firstly, from a theoretical point of view, for all $x, y \in \mathbb{F}_p$, x^y is algebraic. Secondly, the function is bivariate, thus defined on at least 2^{117} inputs in double precision. This makes the computation of $\overline{\varepsilon}$ by current exhaustive research methods impossible [14, 10]. Ziv's iterative process must thus be used for correct rounding. The third and most important point is that there exists a relatively great number of arguments x, y such that x^y is a rounding boundary in some rounding mode. For double precision there are at least 2^{35} such inputs. This makes direct tabulation completely impossible. Those cases are nevertheless relatively rare with regard to the whole definition domain of the function. They are not related to another in some simple, say linear, way.

Filtering out inputs x, y with images x^y on rounding boundaries, i.e. floating-point numbers or mid-points between them, is necessary not only for guaranteeing the termination of the correct rounding process. If the IEEE 754 inexact bit is to be set correctly, filtering cases where x^y is a floating-point number in the target format, i.e. an exact result, is necessary.

As far as known to the authors, in addition to our `crlibm` library, for which support of the `power` function is currently being developed, there are two libraries offering a correctly rounding `power` function: `MPFR`[2] and Sun's `libmcr`[3]. Both implementations make tests for exact and mid-point cases. The tests are implemented in a relatively different way. Both use relatively costly base operations such as square root extraction, divisions and even integer number tests such as `GMP`'s[4] test for perfect squares.

In this article we will present an approach for a correct rounding of the `power` function. Further and mainly, we will give an algorithm for detecting exact and mid-point cases of x^y reusing an approximation to x^y computed by a first iteration of Ziv's process. The algorithm does not make usage of any special base operations; it is based only on additions, multiplications, shifts and tests. Inexact cases will be detected and rejected as fast as possible.

The article is organized as follows: in next section 2 some notations are defined. In section 3 our approach for correctly rounding the `power` function is explained. In section 4 the algorithm is given and explained. Its correctness will be proven in section 5. Section 6

[2] `http://www.mpfr.org/`
[3] `http://www.sun.com/download/products.xml?id=41797765`
[4] `http://www.swox.com/gmp/`

shows some performance results. Conclusions can be found in section 7.

2 Notations

In the following, we will denote by $\mathbb{F}_p$ the set of floating-point numbers with unbounded exponent range and a precision of the significant of p bits. In particular, we will denote by $\mathbb{F}_{53}$ the set of the double precision numbers.

We define the rounding operator $\circ_p : \mathbb{R} \to \mathbb{F}_p$ by $\circ_p(\hat{x}) = x \in \mathbb{F}_p$ such that $\forall x' \in \mathbb{F}_p \ . \ |x' - \hat{x}| > |x - \hat{x}|$. Thus, $\circ_p(x)$ denotes the rounding to nearest in precision p. We will denote by $\lfloor x \rceil$ the integer $n \in \mathbb{N}$ nearest to x. The operation $\lfloor \cdot \rceil$ is a rounding and therefore principally subject to the Table Maker's Dilemma.

Remark that for $x, y \in \mathbb{F}_p$, x^y is an exact or mid-way case for $\mathbb{F}_p$ iff $x^y \in \mathbb{F}_{p+1}$ holds.

3 A correctly rounded `power` function

Arguments $(x, y) \in \mathbb{F}_{54}^2$ for exact or mid-point cases $x^y \in \mathbb{F}_{54}$ of the double precision `power` function cannot be simply tested. The reason is that they are since not trivially distributed in $\mathbb{F}_{53}^2$ and are of great number. They are nevertheless relatively rare in comparison with the whole set of valid inputs to the function.

In fact, a simple estimate shows that there are about 2^{117} inputs to the power function in double precision. There are at least 2^{26} perfect squares that can be written on 53 bits significants and at least 2^9 correspondant exponents. Hence at least 2^{35} nontrivial exact inputs to the `power` function are exact cases. Nevertheless, as exact cases must be representable on at most 54 bits and 2^{10} exponents, there cannot be more than 2^{64} of them. The probability pr_1 of an exact case is thus less than $pr_1 \leq 2^{64-117} = 2^{-51}$, assuming random distribution of the inputs.

In order to optimize average performance, it is useful to start Ziv's iteration process with a first approximation $p = x^y \cdot (1 + \varepsilon)$ and to try to reject as many as possible inexact cases by a simple rounding test [5] on the approximation p. Exact, midpoint and hard-to-round cases behave similar with respect to a rounding test comparing the result to a $\frac{1}{2} \cdot$ ulp value. If the approximation is not sufficiently precise then the relatively costly check for exact and mid-point cases must be executed. However this is the only case for its execution.

The average execution time t in such an approach is $t = t_1 + (pr_1 + pr_2) \cdot (t_2 + t_3)$ where t_1 is the time needed for the first approximation, t_2 is the time for the exact case check, t_3 is the time for the higher Ziv process iterations and where pr_1 is the probability for an exact case and pr_2 the one for a hard-to-round case. Supposing random distribution of the bits of the infinite precision significant, $pr_1 + pr_2$ is proportional to $\hat{\varepsilon} \cdot 2^p$ where $\hat{\varepsilon}$ is the bound for the relative error ε of the first approximate, $|\varepsilon| \leq \hat{\varepsilon}$, and p is the target precision to be rounded to [5]. For double precision with $\hat{\varepsilon} = 2^{-60}$ and $p = 53$, $pr_1 + pr_2 \approx \frac{1}{100}$.

In contrast, in an approach where exact cases are filtered before approximating the function the average execution time will be $t = t_2 + (1 - pr_1) \cdot (t_1 + pr_2 \cdot t_3)$. Since $pr_1 \approx 2^{-51}$ and t_2 is not negligible with regard to t_1, the first approach is more performant.

For this reason, our algorithm for the `power` function implements the approximate-and-test approach. The `power` function in the `MPFR` library follows this principle, too. The library `libmcr` uses the other approach where input are first checked for exactness.

The probability $pr_2 \approx \frac{1}{100}$ of hard-to-round cases that are worse than the first approximation accuracy $\hat{\varepsilon} = 2^{-60}$, is much higher than the probability of an exact case. The exact case detection algorithm should thus try to reuse the intermediate result $p = x^y \cdot (1 + \varepsilon)$ in order to reject as fast as possible the inexact, hard-to-round cases.

All exact and mid-point cases for double-precision are in $\mathbb{F}_{54}$. One observes that the rounding (in any rounding mode) to double precision, i.e. 53 bits, and the rounding to nearest to a 54 bit format cannot be subject to the Table Maker's Dilemma simultaneously. In fact, if the Table Maker's Dilemma occurs on the rounding $\circ_{53}(\hat{x})$, the infinite precision significant of $\hat{x}$ has a long series of consecutive zeros or one starting at the 55th bit. If the Table Maker's Dilemma simultaneously occured on $\circ_{54}(\hat{x})$, there would be a zero or one at the 55th bit and followed by a series of ones respectively zeros starting at the 56th bit [12]. The first approximation is accurate to at least 2^{-60} in relative error. The rounding $\circ_{54}(x^y)$ can thus be computed correctly on all inputs where the rounding test checking for table maker's dilemma cases rejects rounding to 53 bits.

Hence the exact and mid-point test can use the information $\circ_{54}(x^y)$. Since $x^y = \circ_{54}(x^y)$ iff $x^y \in \mathbb{F}_{54}$, the test whether $x^y \in \mathbb{F}_{54}$ reduces to testing if $\circ_{54}(p) = \circ_{54}(x^y)$ is equal to x^y.

In practice, this additional information allows for very efficient filtering of inexact cases as will be shown below. The `power` implementation in `MPFR` nevertheless does not use it. This is one of the main contributions of this work.

4 Detecting exact and mid-point rounding cases

The algorithm for detecting exact and mid-point rounding cases must check whether for given $x, y \in \mathbb{F}_{53}$ and $z = \circ_{54}(x^y) \in \mathbb{F}_{54}$, $x^y = z$. For negative y the test becomes simple because only rationals with integer powers of 2 in the denominator can be represented in floating-point.

For positive y, defining $x = 2^E \cdot m$, $y = 2^F \cdot n$, $z = 2^G \cdot k$, $E, F, G \in \mathbb{Z}$, $m, n, k \in \mathbb{N}$, the test $x^y = z$ becomes equivalent to showing that $2^{E \cdot 2^F \cdot n} \cdot m^{2^F \cdot n} = 2^G \cdot k$. When assuming additionally that m, n and k are odd, the test can be decomposed into two tests, namely $E \cdot 2^F \cdot n = G$ and $m^{2^F \cdot n} = k$ (cf. section 5, lemma 5.1).

The test $E \cdot 2^F \cdot n = G$, i.e. $E \cdot y = G$ is not very costly. In contrast, it rejects a high number of hard-to-round inexact results that have not been filtered by the rounding test. Hard-to-round cases with a final rounding error of less than 2^{-60}, such that $E \cdot y$ is integer, are already relatively rare. G is not the IEEE 754 exponent of $z = \circ_{54}(x^y)$ but a particular exponent value such that $k = 2^{-G} \cdot z$ is odd. By this reason the test is an effective filter for most hard-to-round inexact cases because $E \cdot y = G$ becomes extremly unlikely in pratice. Tested on at least 2^{40} random arguments, our algorithm executed the test about 2^{32} times but did not pass it once. We could construct only one example on which x^y is not exact but on which the test passes. It is the input $x = 2^{-928} \cdot 5794621699391487$, $y = 2^{-5} \cdot 33$.

In our algorithm, the following test for $m^{2^F \cdot n} = k$ gets executed very rarely (with a probability of $\approx 2^{-80}$) on random input. Both `MPFR` and `libmcr` do not take advantage of this simple average performance improvement chance.

The second test for $m^{2^F \cdot n} = k$ decomposes into two main cases depending on the sign of F. If F is positive or 0, $2^F \cdot n$ is integer and can be shown to be bounded by a small value (cf. section 5, lemma 5.2). The value $m^{2^F \cdot n}$ can be exactly computed by repeated multiplication or repeated squaring and multiplying relatively easily and in a performant way.

If F is negative, it can be shown that for $m, n, k \in 2\mathbb{N} + 1$, the egality $m^{2^F \cdot n} = k$ holds

iff $m^{2^F} \in \mathbb{N}$ is satisfied (cf. section 5, lemma 5.3). Testing $m^{2^F \cdot n} = k$ reduces thus to testing if there exists $j \in \mathbb{N}$ such that $j = m^{2^F}$, computing j and testing $j^n = k$. The last test is equivalent to testing $m^{2^F \cdot n} = k$ for $F = 0$ and $m = j$; this test has been explained above.

Different ways exist to perform the basic test whether m is of the form j^{2^i}, $i, j \in \mathbb{N}$, i.e. a multiple or higher perfect square.

4.1 Testing if an integer m is of the form j^{2^i}, $i, j \in \mathbb{N}$

Both libraries `MPFR` and `libmcr` perform the test $m^{2^F} \in \mathbb{N}$ respectively their equivalent form of the test, by repeated square root extraction followed by additional tests. They either check at each step that no rounding error occured on square root extraction or test if a number is a perfect square. This is done by different means: testing if the inexact flag has not been set during the operation, checking conditions on a double-length mantissa approximating $\sqrt{m_i}$. The test if the given number is a perfect square is done by number theoretical algorithms implying divisions and remainders. All these operations are extremely expensive.

From a less number theoretical point of view, the test $m^{2^F} \in \mathbb{N}$ means computing $j' = m^{2^F}$ and testing if the result j' is an integer. Since m^{2^F} can only be approximated, it seems to be impossible. Nevertheless if it is possible to compute the integer j nearest to m^{2^F}, i.e. m^{2^F} itself if it is integer, and to perform the inverse operation $j^{2^{-F}}$ exactly, the test reduces to $\left(\left\lfloor m^{2^F} \right\rceil\right)^{2^{-F}} = m$. It means that the algorithm implicitely tests whether the rounding error $m^{2^F} - \left\lfloor m^{2^F} \right\rceil$ has been equal to 0.

The values F and m can be bounded by very small values. The inverse operation $j^{2^{-F}}$ can, thus, be performed exactly by repeated squaring. It can be shown that if one of the intermediate results cannot be hold on a target precision floating-point variable, m^{2^F} cannot be integer. The Dekker sequence for exact multiplication can easily be adapted for this kind of operation [7].

Computing $\left\lfloor m^{2^F} \right\rceil$ efficiently seems to be impossible, too. On the operation $j = \lfloor j' \rceil = \left\lfloor m^{2^F} \right\rceil$ the Table Maker's Dilemma may occur. This is, nevertheless, of no importance in this case. It does not occur on numbers $m^{2^F} \in \mathbb{N}$ because $\lfloor \cdot \rceil$ is a fixed point rounding to the nearest. If it occured, the fractional part of m^{2^F} would be near to $\frac{1}{2}$, so far from 0 as for integer m^{2^F}. If it occurs on other values, the rounding error does not become 0, however, and the algorithm returns $false$ anyway.

In order to guarantee that $j = \lfloor j' \rceil$ is correctly rounded for values $m^{2^F} \in \mathbb{N}$, the relative error ε of the approximation of $j = m^{2^F}$, $j' = m^{2^F} \cdot (1 + \varepsilon)$ must just be bounded by approximately half the machine error of the target precision. For all target precisions p (here $p = 53$) m is an integer bounded by $2^p - 1$. The value m^{2^F} is, therefore, bounded by $\sqrt{2^p - 1}$ because $F \leq -1$. The egality $\lfloor n + \delta \rceil = \lfloor n \rceil$ holds for $n \in \mathbb{N}$ if the absolute error $\delta = m^{2^F} \cdot \varepsilon$ is bounded by less than $\frac{1}{2}$. Hence roughly, the following bound must be satisfied: $|\varepsilon| \leq 2^{-2} \cdot \frac{1}{\sqrt{2^p - 1}} \approx 2^{\frac{-p}{2} - 2}$.

The operation $j' = m^{2^F} \cdot (1 + \varepsilon)$ can be performed by standard techniques used in elementary functions. In fact, m^{2^F} can be computed as $m^{2^F} = 2^{2^F \cdot \log_2(m)}$. Since $x = 2^E \cdot m$, $\log_2(m) = \log_2(x) - E$. In our two step approach in which x^y is first approximated before exact and midpoint cases are detected, x^y will generally be computed approximately using

$x^y = 2^{y \cdot \log_2(x)}$. So a very good approximation to $\log_2(x)$ will already be available. The possible cancellation in $\log_2(m) = \log_2(x) - E$ is harmless since E is bounded roughly by the exponent range of the target format. Computing $j' = m^{2^F}$ reduces to a substraction, a multiplication by an integer power of 2 and approximating the function 2^ξ on a relatively small range with an accuracy of about half the target precision. For example, in our implementation, on highly pipelined machines like Intel's Itanium system, 2^ξ is available after about 6 floating point latencies. One square root is available after about 5 latencies [3] whereas up to 5 repeated would be needed. This reinforces the argument for not using repeated square root extraction.

Let us give now our algorithm for testing cases $m^{2^F} \in \mathbb{N}$. The bounds given in the preconditions of the algorithm will be shown in section 5 at lemma 5.2.

Input: $m \in (2\mathbb{N}+1) \cap \mathbb{F}_{53}$, $m \geq 3$
$F \in \mathbb{Z}$, $-5 \leq F \leq -1$
Output: $(p, j) = \begin{cases} (true, m^{2^F}) & \text{if } m^{2^F} \in \mathbb{N} \\ (false, 0) & \text{otherwise} \end{cases}$

1 $j' \leftarrow m^{2^F} \cdot (1+\varepsilon)$ where $|\varepsilon| \leq 2^{-30}$;
2 $j \leftarrow \lfloor j' \rceil$;
3 $i \leftarrow -F$;
4 $s \leftarrow j$;
5 **while** $i > 0$ **do**
6 $\quad t_h \leftarrow \circ_{53}(s \cdot s)$; $t_l \leftarrow s \cdot s - t_l$;
7 $\quad$ **if** $t_l \neq 0$ **then return** $(false, 0)$;
8 $\quad s \leftarrow t_h$;
9 $\quad i \leftarrow i - 1$;
10 **end**
11 **if** $s = m$ **then return** $(true, j)$ **else return** $(false, 0)$;

Algorithm 1: **isAHigherSquare**

4.2 The detection algorithm

Let us give now our detection algorithm as presented above. It can be implemented using only double precision arithmetic. Decomposition of numbers uses some bit manipulations. Exact multiplication at line 22 uses the well-known Dekker sequence [7]. The correctness of

the algorithm will be proven in the next section 5.

Input: $x \in \mathbb{F}_{53}, x > 0, y \in \mathbb{F}_{53}, y \neq 0, y \neq 1$
$H \in \mathbb{Z}, k_h, k_l \in \mathbb{F}_{53}$ such that $2^H \cdot (k_h + k_l) = \circ_{54}(x^y)$
Output: a predicate $P(x, y) = (x^y \in \mathbb{F}_{54})$

if $y < 0$ **then**
  if $\neg(\exists a \in \mathbb{Z} \,.\, 2^a = x)$ **then return** *false*;
  $a \leftarrow \log_2(x)$;
  if $a \cdot y \notin \mathbb{Z}$ **then return** *false* **else return** *true*;
else
  Let $E, F, G \in \mathbb{Z}$, $m, n \in (2\mathbb{N}+1) \cap \mathbb{F}_{53}$, $r \in \mathbb{N} \cap \mathbb{F}_{53}$ such that
    $2^E \cdot m \leftarrow x$; $2^F \cdot n \leftarrow y$; $2^G \cdot (2 \cdot r + 1) \leftarrow 2^H \cdot (k_h + k_l)$;
  if $E \cdot y \neq G$ **then return** *false*;
  if $m = 1$ **then**
    if $r = 0$ **then return** *true* **else return** *false*;
  end
  if $F < 0$ **then**
    if $F < -5$ **then return** *false*;
    $(p, j) \leftarrow$ **isAHigherSquare**(F, m);
    if $p = false$ **then return** *false*;
    $m \leftarrow j$; $F \leftarrow 0$;
  end
  if $2^F \cdot n > 35$ **then return** *false*;
  $z \leftarrow 1$;
  $t \leftarrow 2^F \cdot n - 1$;
  while $t > 0$ **do**
    $p_h \leftarrow \circ_{53}(z \cdot m)$; $p_l \leftarrow z \cdot m - p_h$;
    if $p_l \neq 0$ **then return** *false*;
    $z \leftarrow p_h$;
    $t \leftarrow t - 1$;
  end
  $z_h \leftarrow \circ_{53}(z \cdot m)$; $z_l \leftarrow z \cdot m - z_h$;
  if $z_h + z_l \neq 2 \cdot r + 1$ **then return** *false* **else return** *true*;
end

Algorithm 2: **detectExactCase**

5 Correctness proofs of the algorithms

The correctness of our algorithms 1 **isAHigherSquare** and 2 **detectExactCase** can be shown as follows. We establish the fact that the test $x^y = z$ decomposes into two tests:

Lemma 5.1
Assume that $E, F, G \in \mathbb{Z}$, $m, n, k \in 2\mathbb{N}+1$. *Thus the following holds*

$$2^{E \cdot 2^F \cdot n} \cdot m^{2^F \cdot n} = 2^G \cdot k \quad \Leftrightarrow \quad \begin{array}{l} E \cdot 2^F \cdot n = G \\ \wedge \quad m^{2^F \cdot n} = k \end{array}$$

Proof (sketch)
Consider the fact that $m^{2^F \cdot}$ and k or m^n and $k^{2^{-F}}$ are odd integers and $E \cdot 2^F \cdot n - G \neq 0$ or $E \cdot n - G \cdot 2^{-F} \neq 0$. Thus the equations $2^{E \cdot 2^F \cdot n - G} \cdot m^{2^F \cdot n} = k$ or $2^{E \cdot n - G \cdot 2^{-F}} \cdot m^n = k^{2^{-F}}$ yield to contradictions because their left-hand sides are even integers and their right-hand sides are odd. ∎

The algorithms check several bounding conditions on the inputs. These bounds can be shown as follows.

Lemma 5.2
Let $m \in 2\mathbb{N}+1$ be bounded by $3 \leq m \leq 2^{53}-1$. Let $n \in 2\mathbb{N}+1$ be bounded by $1 \leq n \leq 2^{53}-1$. Let $r \in \mathbb{N}$ be bounded by $0 \leq r \leq 2^{53}-1$. Let $F \in \mathbb{Z}$ be an integer. Assume that $m^{2^F \cdot n} = 2 \cdot r + 1$. Thus $2^F \cdot n$ is bounded by $2^F \cdot n \leq 35$ and F is bounded by $-5 \leq F \leq 5$.

Proof
In the first place let us show the upper bounds. Since $r \leq 2^{53}-1$, we know that $2 \cdot r + 1 \leq 2^{54}$. Thus, $m^{2^F \cdot n} \leq 2^{54}$ and $2^F \cdot n \cdot \log_2(m) \leq 54$. Since $m \geq 3$, we have $\log_2(m) \geq \log_2(3) > 0$. Hence, $2^F \cdot n \leq \frac{54}{\log_2(m)} \leq \frac{54}{\log_2(3)} < 34.08 < 35$. This is the upper bound to be shown for $2^F \cdot n$. Since $n \geq 1$, we have $2^F \leq 35$ and therefore $F \leq 5.13$. Since F is integer, we have the given bound $F \leq 5$.

Let us show now that $F \geq -5$. Without loss of generality, we can suppose that F is negative. Let p_i, q_i be prime numbers such that $i \neq i' \Rightarrow p_i \neq p_{i'} \wedge q_i \neq q_{i'}$. Let $\alpha_i, \beta_i \in \mathbb{N} \backslash \{0\}$ be valuations such that $m = \prod_i p_i^{\alpha_i}$ and $2 \cdot r + 1 = \prod_i q_i^{\beta_i}$ Since $F \leq -1$, 2^{-F} is integer. Thus we have $m^n = (2 \cdot r + 1)^{2^{-F}}$ where m^n and $(2 \cdot r + 1)^{2^{-F}}$ are integers. In consequence the egality $\prod_i p_i^{\alpha_i \cdot n} = \prod_i q_i^{\beta_i \cdot 2^{-F}}$ holds and there exists a permutation σ such that $\forall i \,.\, p_i = q_{\sigma(i)} \wedge \alpha_i \cdot n = \beta_{\sigma(i)} \cdot 2^{-F}$. Since m is an odd integer and $m \geq 3$, $\forall i \,.\, p_i \geq 3$ holds. Further $m \leq 2^{53} - 1$ and therefore $3^{\alpha_i} \leq 2^{53}$ and $\alpha_i \leq 53 \cdot \frac{\log(2)}{\log(3)} \leq 34$. Let $\kappa_i \in 2\mathbb{N} + 1$ be odd integers and $\gamma_i \in \mathbb{N}$ valuations such that $\forall i \,.\, \alpha_i = 2^{\gamma_i} \cdot \kappa_i$. Such κ_i and γ_i exist for all α_i because $\alpha_i \in \mathbb{N}$. Since $\forall i \,.\, \alpha_i \geq 1$, the following upper bounds are satisfied: $\forall i \,.\, \kappa_i \geq 1$. As $\forall i \,.\, \alpha_i \leq 34$, the following holds $2^{\gamma_i} \leq 34$ and $\gamma_i \leq \log_2(34) \leq 5.09$ Since $\gamma_i \in \mathbb{N}$, $\gamma_i \leq 5$. The following identity has been shown to hold: $2^{\gamma_i} \cdot (\kappa_i \cdot n) = \beta_{\sigma(i)} \cdot 2^{-F}$ Since n and all κ_i are odd integers, $\kappa_i \cdot n$ is odd. Further, $\beta_{\sigma(i)}$ is an integer. Thus, $-F$ is upper bounded by γ_i which is bounded by 5. So $F \geq -\gamma_i \geq -5$. ∎

Remark that the upper bounds given for $2^F \cdot n$ and $|F|$ are slowly increasing functions of the target precision p. Actually, $2^F \cdot n = \mathcal{O}(p)$ and $|F| = \mathcal{O}(\log p)$. The worst-case complexity and run-time of algorithms 1 and 2 **isAHigherSquare** and **detectExactCases** are linear functions in these bounds. Therefore our algorithms can be extended for higher precision target formats like the = 64 bit double-extended and $p = 112$ bit quad precision. For example, for $p = 112$, $2^F \cdot n$ will be bounded roughly by $2^F \cdot n \leq 71$ and $|F| \leq 6$.

The test $m^{2^F \cdot n} = k$ can be decomposed into two tests, $m^{2^F} = j \in \mathbb{N}$ and $j^n = k$:

Lemma 5.3
Assume $m, n, k \in 2\mathbb{N} + 1$, $F \in \mathbb{Z}$, $F \leq -1$.
Thus $m^{2^F \cdot n} = k \Leftrightarrow \exists j \in \mathbb{N} \,.\, \left(j = m^{2^F} \wedge j^n = k\right)$ holds.

Proof (sketch)
It suffices to remark that a 2^{-F}-root of an integer m is integer only if all valuations of the prime factor decomposition of m are divisible by 2^{-F} and that n is odd. So, if $j = m^{2^F}$ is not integer, there exists a valuation in the prime decomposition as well of m as of m^n that is not divisible by 2^{-F} but all valuations of $k^{2^{-F}}$ are divisible by 2^{-F}. ∎

We are now able to give the correctness proof for our algorithm 1 **isAHigherSquare**.

Theorem 5.4
Algorithm 1 ***isAHigherSquare*** *is correct.*
It means $\forall m \in \mathbb{F}_{53} \cap (2\mathbb{N}+1)\ .\ 3 \leq m \leq 2^{53}-1$ *and* $\forall F \in \mathbb{Z}\ .\ -5 \leq F \leq -1$ *the algorithm terminates and returns* $\left(true, m^{2^F}\right)$ *if* $m^{2^F} \in \mathbb{N}$ *and* $(false, 0)$ *otherwise.*

Proof (sketch)
Showing termination is not a problem. For proving correctness, it suffices to remark that the integer j is computed without occurence of the Table Maker's Dilemma for $m^{2^F} \in \mathbb{N}$ because j' is upper bounded by $2^{27} \geq \sqrt{2^{53}-1}$ and exact to at least 30 bits. Further, since j is integer and $j^{2^{-F}} = m$ is upper bounded by $2^{53}-1$, all j^{2^i}, $0 \leq i \leq -F$, are representable on 53 bits. Thus, the algorithm does not return $(false, 0)$ at line 7 and the loop computes $s = j^{2^{-F}} = m$ exactly. The other way round, if $m^{2^{-F}} \notin \mathbb{N}$, there is a nonzero rounding error on line 2 upon computing $\lfloor j' \rceil$. So if by chance, all intermediate squarings at line 6 are exact, $j^{2^{-F}} = s \neq m$. Hence, the algorithm returns $(false, 0)$. ∎

The correctness of our algorithm 2 **detectExactCases** will be shown also for negative y that we can classify as follows.

Lemma 5.5
Assume that $x, y \in \mathbb{F}_{53}$ *such that* $x > 0$, $y < 0$ *and* $\circ_{53}(x^y) \notin \{\mathrm{NaN}, +\infty, -\infty\}$.
Thus $x^y \in \mathbb{F}_{54}$ *iff* $\exists a \in \mathbb{Z}\ .\ (2^a = x \wedge a \cdot y \in \mathbb{Z})$.

Proof
The existence of the indicated a clearly implies $x^y \in \mathbb{F}_{54}$. The other implication can be as follows: Assume that $x^y \in \mathbb{F}_{54}$ but the contrary of the property to be implied. Since $x, y \in \mathbb{F}_{53}$ and $x^y \in \mathbb{F}_{54}$, there exist odd integers $m, n, k \in 2\mathbb{N}+1$ and signed integers $E, F, G \in \mathbb{Z}$ such that $x = 2^E \cdot m$, $y = -2^F \cdot n$ and $x^y = 2^G \cdot k$. This yields to $m^{2^F \cdot n} = 2^{-G-E \cdot 2^F \cdot n} \cdot \frac{1}{k}$. There are two case depending on the sign of F. If $F \geq 0$ then 2^F is an integer, $2^F \cdot n$ is an integer and $-G - E \cdot 2^F \cdot n$ is a signed integer. There exists therefore an integer $a \in \mathbb{N}$, a signed integer $b \in \mathbb{Z}$ and an odd integer $c \in 2\mathbb{N}+1$ such that $m^a = 2^b \cdot \frac{1}{c}$ by the definitions $a = 2^F \cdot n$, $b = -G - E \cdot 2^F \cdot n$ and $c = k$. If $F < 0$ then 2^{-F} is integer, $-G \cdot 2^{-F}$, $E \cdot n$ and $-G \cdot 2^{-F} - E \cdot n$ are signed integers and $k^{2^{-F}}$ is an odd integer. So there exist an integer $a \in \mathbb{N}$, a signed integer $b \in \mathbb{Z}$ and an odd integer $c \in 2\mathbb{N}+1$ such that $m^a = 2^b \cdot \frac{1}{c}$ by $a = n$, $b = -G \cdot 2^{-F} - E \cdot n$ and $c = k^{2^{-F}}$.

If $b \geq 0$, 2^b is integer. Since m^a is integer, $\frac{2^b}{c}$ is integer. As c is odd, $\gcd\left(2^b, c\right) = 1$. In consequence, c is equal to 1, $c = 1$. If $b < 0$, 2^{-b} and $2^{-b} \cdot c$ are integer. Since m^a is integer, $\frac{1}{2^{-b} \cdot c}$ is integer. Hence, $2^{-b} \cdot c = 1$ and thus, $c = 1$ because c is odd and 2^{-b} is integer.

Since $c = 1$ and $c = k$ or $c = k^{2^{-F}}$, k is equal to 1 in all cases. This implies that $x^y = 2^G \cdot 1$ is an integer power of 2. In consequence, $x = 2^{\frac{G}{y}}$. Since 2^ξ is transcendental for all algebraic

ξ that are not signed integers (see [9]), and since x is algebraic, there exists an $a = \frac{G}{y} \in \mathbb{Z}$. Thus, $a \cdot y = G \in \mathbb{Z}$ and $x = 2^{\frac{G}{y}} = 2^{\frac{a \cdot y}{y}} = 2^a$. This yields a contradiction with the hypotheses. ∎

Here is finally the correctness proof of our algorithm for detecting exact cases.

Theorem 5.6
Algorithm 2 ***detectExactCase*** *is correct.*
This means $\forall x \in \mathbb{F}_{53}, x > 0$ *and* $\forall y \in \mathbb{F}_{53}, y \neq 0, y \neq 1$ *such that* $2^{-1075} \leq x^y \leq 2^{1024}$ *and* $\forall k_h, k_l \in \mathbb{F}_{53}$, $\forall H \in \mathbb{Z}$ *such that* $2^H \cdot (k_h + k_l) = \circ_{54}(x^y)$ *the algorithm returns true iff* $x^y \in \mathbb{F}_{54}$.

Proof (sketch)
Showing termination is not a problem. The case where $y < 0$ can be proven by direct application of lemma 5.5. Otherwise consider the following: after the execution of line 27, $z_h + z_l = \widetilde{m}^{2^{\widetilde{F}} \cdot n}$ where $\widetilde{m}$ and $\widetilde{F}$ are the values of the variables m and F taken at line 18. The value $\widetilde{m}$ is an integer bounded by $2^{53} - 1$ such that $\widetilde{m}^i \leq 2^{53} - 1$ for $0 \leq i \leq 2^{\widetilde{F}} \cdot n - 1$ if $x^y \in \mathbb{F}_{54}$ by 5.2. So for x^y, at each iteration of the loop, $z \cdot m = \widetilde{m}^i \in \mathbb{F}_{53}$. Since $j = m^{2^F}$ by theorem 5.4 if the control flow goes through line 14, $z_h + z_l = m^{2^F \cdot n}$ where m, F and n are the values computed at lines 6 and 7. Further, $E \cdot y = G$, i.e. $E \cdot 2^F \cdot n = G$. Thus, by lemmata 5.1, 5.2 and 5.3, $x^y = z_h + z_l$ and since $z_h + z_l = 2 \cdot r + 1 \in \mathbb{F}_{54}$ if the algorithm returns *true*, $x^y \in \mathbb{F}_{54}$ in this case. If $x^y \notin \mathbb{F}_{54}$ the algorithm returns either $false$ at one of the lines 8, 13, 15, 18 and 23 or $z_h + z_l = x^y$ may not be equal to $2 \cdot r + 1$ because $2 \cdot r + 1 \in \mathbb{F}_{54}$. So it returns $false$ in any case. The special case $m = 1$ is trivial. ∎

6 Performance results

The following comparative performance results of our `power` function in `crlibm` and of the function in Sun's `libmcr` have been observed. Timing measurements have been obtained on a Intel Xeon 2.40GHz processor running on Debian Linux 2.6.10-server when compiling both libraries `crlibm` and `libmcr` with `gcc` version 3.3.5.

On random values $x \in \mathbb{F}_{53}$, $y \in \mathbb{F}_{53}$ such that no infinity or NaN and no wrong rounding occurs, the `crlibm` outperforms `libmcr` on average with 1746 time units with regard to 3235 time units. On maximum timings it is slightly less performant with 8456 time units in comparison with `libmcr`'s 6388 time units. These maximum timings are mainly caused by costly subnormal rounding of x^y in its gradual underflow range. Additional analysis shows that on not subnormal results that are not exact or midway cases, the result is available on average after 1200 time units using `crlibm` and after only 2900 time units using `libmcr`. This is a speed-up on average of a factor 2 in the favor of our `power` function.

On values $x \in \mathbb{F}_{53}$, $y \in \mathbb{F}_{53}$ such that $x^y \in \mathbb{F}_{54}$ without occurence of infinity, NaN or subnormal rounding, Sun's `libmcr` is faster than our `crlibm` on average and on maximum timings with 1103 time units versus 1747 units respectively 1688 time units versus 2032 time units. This speed-down of `crlibm` of maximally 68% must be seen relatively to the fact that `libmcr` effectuates an exact and mid-point case check on all non-trivial inputs before approximating x^y with a first intermediate precision and `crlibm` first approximates the function. The check implemented in `libmcr` computes the value x^y simultaneously while checking if it is exact or mid-point. This allows `libmcr` to return x^y if it is exact after 1103

cycles on average. However it costs the library's function the same cost when returning $\circ(x^y)$ if the case is not exact. Since the average timing on inexact results is 2900 time units, about 38% are spent for the exact case test *at each call*. In contrast, since an inexact case result is available after 1200 time units on average with our algorithm and the exact case result after the test is available after about 1750 time units, only about 31% are spent on testing the exactness of a case on average, *if the case is exact*.

7 Conclusions and future work

We have given an approach for implementing a correctly rounded `power` function, $\circ(x^y)$, and an algorithm for detecting exact and mid-point rounding cases of this function in double precision. The correctness of the algorithm is completely proven. It can be implemented using only double precision arithmetic. It does not make usage of any costly operations such as divisions, remainders or square root extraction. It follows the common principle of returning fast answers on frequent cases and allows for reusing maximally already computed approximations. Nevertheless, its worst case timings are bounded by practical observation as well as by theoretical considerations.

Our algorithm perfectly adapts and scales to higher precisions, like e.g. double-extended or quad precision. Its complexity and practical timings are linear in the target precision.

Performance results show that the proposed approach can speed up the average case performance of the `power` function by a factor of about 1.8 inducing an additional cost of only 31% on exact cases.

The given algorithm is optimized for rejecting as fast as possible all inexact values. Nevertheless, some of its control flow paths seem to be executed very rarely or even no time in practice. They are currently necessary for permitting a complete correctness proof. This proof and in consequence the algorithm might be optimized still in the future.

References

[1] CR-Libm, a library of correctly rounded elementary functions in double-precision. http://lipforge.ens-lyon.fr/www/crlibm/.

[2] ANSI/IEEE. Standard 754-1985 for binary floating-point arithmetic, 1985.

[3] M. Cornea, J. Harrison, and P.T.P Tang. *Scientific Computing on Itanium-based Systems*. Intel Press, 2002.

[4] F. de Dinechin, D. Defour, and C. Lauter. Fast correct rounding of elementary functions in double precision using double-extended arithmetic. Technical Report 2004-10, LIP, École Normale Supérieure de Lyon, March 2004.

[5] F. de Dinechin, A. Ershov, and N. Gast. Towards the post-ultimate libm. In *17th IEEE Symposium on Computer Arithmetic*, Cape Cod, Massachussets, June 2005.

[6] F. de Dinechin, Ch. Q. Lauter, and J.-M. Muller. Fast and correctly rounded logarithms in double-precision. *Theoretical Informatics and Applications*, 2006. to appear.

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

[8] Shmuel Gal and Boris Bachelis. An accurate elementary mathematical library for the IEEE floating point standard. *ACM Transactions on Mathematical Software*, 17(1):26–45, March 1991.

[9] G. H. Hardy and E. M. Wright. *An introduction to the theory of numbers*. Oxford University Press, 1979.

[10] V. Lefèvre and J.-M. Muller. Worst cases for correct rounding of the elementary functions in double precision. In Neil Burgess and Luigi Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pages 111–118, Vail, Colorado, 2001. IEEE Computer Society Press, Los Alamitos, CA.

[11] V. Lefèvre, J.-M. Muller, and A. Tisserand. The table maker's dilemma. Technical Report RR1998-12, Laboratoire de l'Informatique du Parallélisme, Lyon, France, 1998.

[12] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhauser, Boston, 1997.

[13] Y. V. Nesterenko and M. Waldschmidt. On the approximation of values of exponential and logarithm by algebraic numbers (in russian). *Math. Zapiski*, (2):23–42, 1996.

[14] D. Stehlé, V. Lefèvre, and P. Zimmermann. Worst cases and lattice reduction. In Jean-Claude Bajard and Michael Schulte, editors, *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, pages 142–147, Santiago de Compostela, Spain, 2003. IEEE Computer Society Press, Los Alamitos, CA.

[15] A. Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, September 1991.