Comparison between binary64 and decimal64 floating-point numbers

Nicolas Brisebarre, Marc Mezzarobba, Jean-Michel Muller, Christoph Lauter

Comparison between binary64 and decimal64 floating-point numbers

Nicolas Brisebarre, Marc Mezzarobba
Jean-Michel Muller
Laboratoire LIP
CNRS, ENS Lyon, INRIA,
Univ. Claude Bernard Lyon 1
Lyon, France
first-name.last-name@ens-lyon.fr

Christoph Lauter
Université Pierre et Marie Curie
Laboratoire d'Informatique de Paris 6
Paris, France
christoph.lauter@lip6.fr

***Abstract*—We introduce a software-oriented algorithm that allows one to quickly compare a binary64 floating-point (FP) number and a decimal64 FP number, assuming the "binary encoding" of the decimal formats specified by the IEEE 754-2008 standard for FP arithmetic is used. It is a two-step algorithm: a first pass, based on the exponents only, makes it possible to quickly eliminate most cases, then when the first pass does not suffice, a more accurate second pass is required. We provide an implementation of several variants of our algorithm, and compare them.**

I. Introduction

The IEEE 754-2008 Standard for Floating-Point Arithmetic [3] specifies several binary and decimal formats. The "basic interchange formats" of the standard are presented in Table I.

	binary32	binary64	binary128
precision (bits)	24	53	113
$e_{\max}$	$+127$	$+1023$	$+16383$
$e_{\min}$	-126	-1022	-16382
		decimal64	decimal128
precision (digits)		16	34
$e_{\max}$		$+384$	$+6144$
$e_{\min}$		-383	-6143

TABLE I
THE BASIC BINARY AND DECIMAL INTERCHANGE FORMATS SPECIFIED BY THE 754-2008 STANDARD.

Although the standard does not require that a binary and a decimal number can be compared (it says *floating-point data represented in different formats shall be comparable as long as the operands' formats have the same radix*), such comparisons may nevertheless offer several advantages: it is not infrequent to read decimal data from some database and to have to compare it to some binary floating-point number. That comparison may be slightly inaccurate if the decimal number is preliminarily converted to binary, or, respectively, if the binary number is first converted to decimal.

While it does not require comparisons between floating-point numbers of different radices, the IEEE 754-2008 standard does not forbid them for languages or systems. As the technical report on decimal floating-point arithmetic in C [4] is currently a mere draft, compilers supporting decimal floating-point arithmetic handle code sequences such as the following one at their discretion and often in an unsatisfactory way:

```
double x = ...;
_Decimal64 y = ...;
if (x <= y) {
    ...
}
```

As it occurs, this sequence is translated, for example by Intel's icc 12.1.3, into a conversion from binary to decimal followed by a decimal comparison. The compiler emits no warning that the boolean result might not be the expected one because of rounding in the comparison.

This kind of strategy may lead to inconsistencies. Imagine such a "naïve" approach built as follows: when comparing a binary floating-point number x_2 of format $\mathcal{F}_2$, and a decimal floating-point number x_{10} of format $\mathcal{F}_{10}$, we first convert x_{10} to the binary format $\mathcal{F}_2$ (that is, we replace it by the $\mathcal{F}_2$ number nearest x_{10}), and then we perform the comparison in binary. Denote $\lessdot$, $\leqslant$, $\gtrdot$, and $\geqslant$ as the comparison operators so defined. Consider the following variables (all exactly represented in their respective formats):

- $x = 3602879701896397/2^{55}$, declared as a binary64 number;
- $y = 13421773/2^{27}$, declared as a binary32 number;
- $z = 1/10$, declared as a decimal64 number.

Then it holds that $x \lessdot y$, but also $y \leqslant z$ and $z \leqslant x$. Such an inconsistent result might suffice to prevent a sorting program from terminating.

A direct mixed-radix binary-decimal-comparison might be an answer.

Also, in the long run, allowing "exact" comparison of decimal and binary floating-point numbers will certainly be the only rigorous way of executing program instructions of the form

```
if x > 0.1 then ...
```

unless compilers learn how to round (binary or decimal)

constants figuring in comparisons in a way that does not affect the comparison.

In the following, we aim at introducing an algorithm for comparing a binary64 floating-point number

$$x_2 = M_2 \cdot 2^{e_2-52},$$

and a decimal64 floating-point number

$$x_{10} = M_{10} \cdot 10^{e_{10}-15}.$$

Here M_2 and M_{10} are integers, with $|M_2| \leqslant 2^{53}-1$ and $|M_{10}| \leqslant 10^{16}-1$, and e_2 and e_{10} are the floating-point exponents of x_2 and x_{10}.

Although most of what will be presented in this paper is generalizable to other formats, we restrict ourselves to these two formats, for the sake of simplicity. We will also assume that the so-called *binary encoding* [3], [6] of IEEE 754-2008 is used for the decimal64 format, so that the integer M_{10} is easily accessible in binary. Since comparisons are straightforward if x_2 and x_{10} have different signs or are zero, we assume that $M_2 > 0$ and $M_{10} > 0$. Also, to leave the IEEE 754 flags untouched (unless when required), we will use integer arithmetic rather than floating-point arithmetic to perform our tests.

When x_2 and x_{10} have significantly different orders of magnitude, examining their exponents will suffice to compare them. Hence, we first address the problem of performing a first, exponent-based, test. We then show how to perform the comparison when the first test does not suffice.

II. First step: eliminating the "simple cases" by examining the exponents

Through a possible preliminary binary shift of M_2 (when x_2 is subnormal), we may assume that $2^{52} \leqslant M_2$. This gives the following constraints on e_2 (taking into account that possible shift):

$$-1074 \leqslant e_2 \leqslant 1023. \tag{1}$$

The constraints on e_{10}, read from Table I, are

$$-383 \leqslant e_{10} \leqslant 384. \tag{2}$$

Also, from

$$1 \leqslant M_{10} \leqslant 10^{16}-1,$$

we easily deduce that there exists a unique $\nu \in \{0,1,2,\ldots,53\}$ such that

$$2^{53} \leqslant 2^{\nu} M_{10} \leqslant 2^{54}-1.$$

Hence our initial problem of comparing x_2 and x_{10} reduces to comparing $M_2 \cdot 2^{e_2-52+\nu}$ and $(2^{\nu}M_{10}) \cdot 10^{e_{10}-15}$.

The fact that we "normalize" the decimal significand M_{10} by a binary shift between two consecutive powers of two is of course questionable; M_{10} could also be normalized into the range $10^{15} \leqslant 10^t \cdot M_{10} \leqslant 10^{16}-1$. However, as hardware support for binary encoded decimal floating-point arithmetic is not widespread, a decimal normalization would require a loop multiplying by 10 and testing, whereas the proposed binary normalization can exploit an existing hardware leading-zero counter with straight-line code. The pipeline stalls in the loop are hence avoided.

We hence define

$$\begin{cases} m = M_2, \\ h = \nu + e_2 - e_{10} - 37, \\ n = M_{10} \cdot 2^{\nu}, \\ g = e_{10} - 15, \end{cases} \tag{3}$$

so that

$$\begin{cases} x_2 = m \cdot 2^h \cdot 2^{g-\nu}, \\ x_{10} = n \cdot 5^g \cdot 2^{g-\nu}. \end{cases} \tag{4}$$

Our comparison problem becomes:

$$\textit{Compare } m \cdot 2^h \textit{ with } n \cdot 5^g.$$

We have

$$\begin{aligned} 2^{52} \leqslant m &\leqslant 2^{53}-1, \\ 2^{53} \leqslant n &\leqslant 2^{54}-1. \end{aligned} \tag{5}$$

From the ranges of e_2, e_{10} and ν, one easily deduces

$$\begin{aligned} -398 \leqslant g &\leqslant 369, \\ -1495 \leqslant h &\leqslant 1422. \end{aligned} \tag{6}$$

Note that once the first step, eliminating comparison cases purely based on the exponents, is complete, we will be able to reduce these domains (see Lemma 1). It makes no sense, for instance, to assume that e_2 is tiny and e_{10} is huge: in such a case, the comparison is straightforward.

Now define two functions φ and ψ by

$$\varphi(h) = \lfloor h \cdot \log_5 2 \rfloor, \qquad \psi(g) = \lfloor g \cdot \log_2 5 \rfloor. \tag{7}$$

The function φ is appropriate to perform the first comparison step, as stated in Property 1 below. The other function will be useful in the sequel. We will propose an efficient and easy-to-implement way to compute these functions φ and ψ.

Property 1. *We have*

$$\begin{aligned} g < \varphi(h) &\Rightarrow x_2 > x_{10}, \\ g > \varphi(h) &\Rightarrow x_2 < x_{10}. \end{aligned}$$

Proof: If $g < \varphi(h)$ then $g \leqslant \varphi(h) - 1$, hence $g \leqslant h \log_5 2 - 1$. This implies that $5^g \leqslant (1/5) \cdot 2^h$, therefore $2^{54} \cdot 5^g \leqslant (4/5) \cdot 2^{52} \cdot 2^h$. As a consequence, $n \cdot 5^g < m \cdot 2^h$. If $g > \varphi(h)$ then $g \geqslant \varphi(h) + 1$, hence $g > h \log_5 2$, so that $5^g > 2^h$. This implies $2^{53} \cdot 5^g > (2^{53}-1) \cdot 2^h$, and hence $n \cdot 5^g > m \cdot 2^h$. ∎

Property 2. *Denoting by $\lfloor \cdot \rceil$ the nearest integer function, let*

$$\begin{aligned} L_{\varphi} &= \lfloor 2^{19} \log_5 2 \rceil = 225799, \\ L_{\psi} &= \lfloor 2^{12} \log_2 5 \rceil = 9511. \end{aligned}$$

For all h in the range (6)*, we have*

$$\varphi(h) = \lfloor h \cdot L_{\varphi} \cdot 2^{-19} \rfloor.$$

Similarly, we have

$$\psi(g) = \lfloor L_{\psi} \cdot g \cdot 2^{-12} \rfloor \qquad \textit{for } |g| \leqslant 204,$$

and in the special case $g = 16q$,

$$\psi(16q) = \lfloor L_\psi \cdot q \cdot 2^{-8} \rfloor \qquad \textit{for } |q| \leqslant 32.$$

The products $L_\varphi \cdot h$ and $L_\psi \cdot g$, for g, h in the indicated ranges, can all be computed exactly in (signed or unsigned) 32-bit integer arithmetic. Computing $\lfloor \xi \cdot 2^{-\beta} \rfloor$ of course reduces to a right-shift by β bits.

Proof: Using Maple, we exhaustively check that

$$\varphi(h) = \lfloor h \cdot L \cdot 2^{-19} \rfloor$$

for $|h| \leqslant 1831$, and that

$$\psi(g) = \lfloor L_\psi \cdot g \cdot 2^{-12} \rfloor$$

when either $|g| \leqslant 204$ or $g = 16q$ with $|q| \leqslant 32$. The corresponding quantities $L_\varphi \cdot h$ and $L_\psi \cdot g$ all fit on at most 29 bits in magnitude, plus one sign bit. ∎

From Properties 1 and 2, we easily derive the following algorithm, that performs the first step, based on the exponents of x_2 and x_{10}.

Algorithm 1. ***First step***

- *compute $h = \nu + e_2 - e_{10} - 37$ and $g = e_{10} - 15$;*
- *compute $\varphi(h) = \lfloor L_\varphi \cdot h \cdot 2^{-19} \rfloor$ using 32-bit integer arithmetic;*
- *if $g < \varphi(h)$ then return "$x_2 > x_{10}$", else if $g > \varphi(h)$ then return "$x_2 < x_{10}$", else perform the second step.*

Note that, when x_{10} admits multiple distinct representations in binary64 format (i.e. when its cohort is non-trivial [3]), the success of the first step may depend on the specific representation passed as input. For instance, both $A = \{M_{10} = 10^{15}, e_{10} = 0\}$ and $B = \{M_{10} = 1, e_{10} = 15\}$ are valid representations of the integer 1. Assume we are trying to compare $x_{10} = 1$ to $x_2 = 2$. Using representation A, we have $\nu = 4$, $h = -32$, and $\varphi(h) = -14 > g = -15$, hence the test from Algorithm 1 shows that $x_{10} < x_2$. In contrast, if x_{10} is given in the form B, we get $\nu = 53$, $\varphi(h) = \varphi(2) = 0 = g$, and the test is inconclusive.

We may quantify the quality of this first filter as follows. We say that Algorithm 1 *succeeds* if it answers "$x_2 > x_{10}$" or "$x_2 < x_{10}$" without proceeding to the second step, and *fails* otherwise. Let X_2 and X_{10} denote the sets of *representations* of positive, finite numbers, respectively in binary64 and in decimal64 format. (In the case of X_2, each number has a single representation.) Assuming zeros, infinities, and NaNs have been handled before, the input of Algorithm 1 may be thought of as a pair $(\xi_2, \xi_{10}) \in X_2 \times X_{10}$.

Proposition 1. *Algorithm 1 succeeds for more than 99.9% of the input pairs $(\xi_2, \xi_{10}) \in X_2 \times X_{10}$.*

Proof: The first step succeeds if and only if $\varphi(h) \neq g$, that is, iff $\varphi(\nu + e_2 - e_{10} - 37) \neq e_{10} - 15$. The value of ν depends only on M_{10}. Knowing ν, the number of possible values of M_{10} is

$$n_\nu = \begin{cases} 2^{53-\nu}, & \nu > 0, \\ 10^{16} - 2^{53}, & \nu = 0. \end{cases}$$

In addition, for each fixed ν, we can easily compute the number k_ν of pairs (e_2, e_{10}) such that $-1022 \leqslant e_2 \leqslant 1023$, $-383 \leqslant e_{10} \leqslant 384$, and $\varphi(h) = g$.

Let X_2^{norm} be the subset of X_2 consisting of normal numbers. The number of pairs (e_2, ξ_{10}) such that $\varphi(h) = g$ and $\xi_2 \in X_2^{\mathrm{norm}}$ is

$$N_1 = \sum_{\nu=0}^{53} n_\nu k_\nu = 14\,292\,575\,372\,220\,927\,484 \leqslant 1.55 \cdot 2^{63}.$$

If now x_2 is subnormal, then $h < -623$ and $\varphi(h) < -268$. A rough bound on the number of $(\xi_2, \xi_{10}) \in (X_2 \backslash X_2^{\mathrm{norm}}) \times X_{10}$ such that $g = \varphi(h)$ is hence

$$N_2 = 2^{52} \cdot 10^{16} \cdot (398 - 268) \leqslant 1.12 \cdot 2^{112}.$$

Thus, the number of elements of $X_2 \times X_{10}$ for which the first step fails is bounded by

$$N = 2^{52} N_1 + N_2 \leqslant 1.691 \cdot 2^{115}.$$

This is to be compared with

$$|X_2 \times X_{10}| = (2^{52} - 1) \cdot 2047 \cdot (10^{16} - 1) \cdot 768 \geqslant 1.66 \cdot 2^{125}.$$

We obtain $N/|X_2 \times X_{10}| \leqslant 10^{-3}$. ∎

As a matter of course, pairs (ξ_2, ξ_{10}) will almost never be equidistributed in practice. Hence the previous estimate should not be interpreted as a *probability* of success of Step 1. It seems more realistic to assume that a well-written numerical algorithm will mostly perform comparisons between numbers which are suspected to be close to each other. For instance, in an iterative algorithm where comparisons are used as part of a convergence test, it is to be expected that most comparisons need to proceed to the second step. Conversely, there are scenarios, e.g., checking for out-of-range data, where the first step should almost always be enough.

III. Second step: a closer look at the significands

In the following we assume that $g = \varphi(h)$ (otherwise, the first step already allowed us to compare x_2 and x_{10}).

Define a function

$$f(h) = 5^{\varphi(h)} \cdot 2^{-h} = 2^{\lfloor h \log_5 2 \rfloor \cdot \log_2 5 - h}.$$

We have

$$\begin{cases} f(h) \cdot n > m \Rightarrow x_{10} > x_2, \\ f(h) \cdot n < m \Rightarrow x_{10} < x_2, \\ f(h) \cdot n = m \Rightarrow x_{10} = x_2. \end{cases} \tag{8}$$

The second test consists in performing this comparison, with $f(h) \cdot n$ replaced by an approximation that is accurate enough not to introduce false results.

Ensuring that an approximate test is indeed equivalent to (8) requires to find a lower bound η on the minimum nonzero value of $|m/n - 5^{\varphi(h)}/2^h|$ that may appear at this stage. We want η to be as tight as possible in order to avoid unduly costly computations when approximating $f(h) \cdot n$. The search space is constrained by the following observations.

Lemma 1. *The equality $g = \varphi(h)$ where g and h are defined by* (3) *can hold only if $-787 \leqslant h \leqslant 716$. Additionally,*

- *if* $n \geqslant 10^{16}$*, then* n *is necessarily even;*
- *if* $h \geqslant 680$*, then* $\nu' = h + \varphi(h) - 971 > 0$ *and* n *is divisible by* $2^{\nu'}$.

Proof: Substituting $e_2 = h + g - \nu + 52$ and $g = \varphi(h)$ into inequality (1) yields

$$-1126 \leqslant e_2 - 52 = h + \varphi(h) - \nu \leqslant 971. \tag{9}$$

Since we have $0 \leqslant \nu \leqslant 53$ and $h \cdot \log_5 2 - 1 \leqslant \varphi(h) \leqslant h \cdot \log_5 2$, this implies

$$-1126 \leqslant h \cdot (1 + \log_5 2) \leqslant 1025.$$

The bound on h follows.

Both conditions on n come from the fact that 2^ν divides n (which we denote $2^\nu \mid n$ in the sequel of the paper) by definition of n. If $n \geqslant 10^{16}$, then $\nu \geqslant 1$ and hence n is even. In general, the last inequality from (9) implies $\nu \geqslant \nu'$. This last bound is nontrivial when $h \geqslant 680$. ∎

We now deal with the problem of finding η. This problem is very close to that, considered by Cornea et al. in [1], of finding the required accuracy for performing correctly rounded conversions between the binary and decimal formats of the IEEE 754-2008 Standard. The constraints on m, n and h follow from inequalities (5) and Lemma 1.

Problem 1. *Find the smallest nonzero value of*

$$d_h(m,n) = \left| \frac{5^{\varphi(h)}}{2^h} - \frac{m}{n} \right|$$

under the following constraints

$$\begin{cases} 2^{52} \leqslant m \leqslant 2^{53} - 1, \\ 2^{53} \leqslant n \leqslant 2^{54} - 1, \\ -787 \leqslant h \leqslant 716, \\ n \text{ is even if } n \geqslant 10^{16}, \\ \text{if } h \geqslant 680, \text{ then } \nu' = h + \varphi(h) - 971 > 0 \\ \qquad\qquad \text{and } 2^{\nu'} \mid n. \end{cases}$$

For each integer $h \in [0, 716]$, a numerator of $d_h(m,n)$ is $\left|5^{\varphi(h)}n - m2^h\right|$ and a denominator is $2^h n$.

If $0 \leqslant h \leqslant 53$ and $d_h(m,n) \neq 0$, we have $|5^{\varphi(h)}n - m2^h| \geqslant 1$, hence $d_h(m,n) \geqslant 1/(2^h n) \geqslant 2^{-107}$. This easy argument actually means that for the range of h where equality cases might occur, the *non-zero* minimum is no less that 2^{-107}.

If $h \geqslant 54$, since $5^{\varphi(h)}$ and 2^h are coprime and $n \leqslant 2^{54} - 1$, we have $d_h(m,n) \neq 0$. Minimizing $d_h(m,n)$ is a classical question related to the theory of continued fractions [2], [5], [7]. We now give a short reminder on the results we need in our setting.

Let $\alpha \in \mathbb{Q}$. We build two finite sequences $(a_i)_{0 \leqslant i \leqslant n}$ and $(r_i)_{0 \leqslant i \leqslant n}$ defined by:

$$\begin{cases} r_0 = \alpha, \\ a_i = \lfloor r_i \rfloor, \\ r_{i+1} = 1/(r_i - a_i) \text{ if } a_i \neq r_i. \end{cases}$$

Note that this process is actually Euclid's algorithm. For all $0 \leqslant i \leqslant n$, the rational number

$$\frac{p_i}{q_i} = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + \cfrac{1}{a_i}}}}$$

is the ith *convergent* of the *continued fraction expansion* of α. We write $\alpha = [a_0; a_1, \cdots, a_n]$. The a_i are the *partial quotients* of the expansion. Note that, if we assume $p_{-1} = 1$, $q_{-1} = 0$, $p_0 = a_0$, $q_0 = 1$, we have $p_{i+1} = a_{i+1}p_i + p_{i-1}$, $q_{i+1} = a_{i+1}q_i + q_{i-1}$, and $\alpha = p_n/q_n$.

First assume that the minimum nonzero value will be less than or equal to 2^{-109}. Then, the integers $2^{52} \leqslant m \leqslant 2^{53} - 1$ and $2^{53} \leqslant n \leqslant 2^{54} - 1$ that we want to determine necessarily satisfy $|5^{\varphi(h)}/2^h - m/n| \leqslant 2^{-109} < 1/(2n^2)$: a classical result (see Theorem 19 of [5] for instance) yields that m/n is a convergent of $5^{\varphi(h)}/2^h$. Hence, we only have to compute, for each $h \in [54, 716]$, the convergents of $5^{\varphi(h)}/2^h$ the denominators of which are smaller than 2^{54}. Moreover, we only consider the convergents p_i/q_i which improve the minimum nonzero value and for which there exists $k \in \mathbb{N}$ such that $m = kp_i$ and $n = kq_i$ satisfy the constraints of Problem 1.

We stress that this pruning trick is valid if and only if one eventually obtains a minimum nonzero value lesser than or equal to 2^{-109}. If not, one can use the techniques presented in the Appendix of [1] to get worst and bad cases for the approximation problem we consider (that is, to get values x_2 and x_{10} that are very near). We have favored a different approach, here, because it leads to a less computationally intensive and mathematically simpler method, which is easier to check either manually or with a proof-assistant. However, if one wishes to generate many cases for which x_2 and x_{10} are very close (typically, for the purpose of testing comparison algorithms), the techniques of [1] become of huge interest. We will take advantage of them in Section VI.

The computation shows that the smallest admissible value of $d_h(m,n)$ for $h \geqslant 0$ is $6.3566\ldots \times 10^{-35} < 2^{-113.59}$, attained for

$$h = 612, \quad \frac{m}{n} = \frac{3521275406171921}{8870461176410409}.$$

We address the case $h \in [-787, 0]$ in a similar way. In this case, a numerator of $d_h(m,n)$ is $\left|2^{-h}n - 5^{-\varphi(h)}m\right|$ and a denominator is $5^{-\varphi(h)}n$. We notice that $5^{-\varphi(h)} \geqslant 2^{54}$ if and only if $h \leqslant -54$. If $-53 \leqslant h \leqslant 0$ and $d_h(m,n) \neq 0$, we have $|2^{-h}n - 5^{-\varphi(h)}m| \geqslant 1$, hence $d_h(m,n) \geqslant 1/(5^{-\varphi(h)}n) \geqslant 2^{-108}$. For $-787 \leqslant h \leqslant -54$, we use the same continued fraction tools as above, leading to the following result.

Theorem 1. *The minimum nonzero value of* $d_h(m,n)$ *under the constraints from Problem 1 is*

$$6.0485\ldots \times 10^{-35} < 2^{-113.67},$$

attained for

$$h = -275, \quad \frac{m}{n} = \frac{4988915232824583}{12364820988483254}.$$

We may thus take $\eta = 2^{-113.7}$.

IV. Inequality testing

As already mentioned, the first step of the full comparison algorithm is straightforwardly implementable in 32-bit integer arithmetic. The second one reduces to comparing m and $f(h)\cdot n$, and we have seen in Section III that it is enough to know $f(h)$ with relative accuracy $2^{-113.67}$. We now discuss several ways to perform this comparison. The main difficulty is to efficiently evaluate $f(h)\cdot n$ with just enough accuracy.

A. Direct method

Perhaps the first thing that comes to mind is to implement the criterion from Equation (8) directly, only with $f(h)$ replaced by a sufficiently accurate approximation, read in a table that is indexed with h. Theorem 2 below implies that it is more than enough to compute the product $f(h)\cdot n$ in 128-bit arithmetic, using a 128-bit approximation of $f(h)$.

By Lemma 1, there are 1 504 values of h to consider, which corresponds to a 23.5 KB table with 128-bit precision entries for $f(h)$. The resulting table is quite large but could possibly be reused, for instance, in conversions between binary and decimal formats. One advantage of this method is that the table value $f(h)$ only depends on h, which is available early in the algorithm flow. Thus, the memory latency for the table access can be hidden behind the first step.

Theorem 2. *Assume that μ approximates $f(h)\cdot n$ with relative accuracy $\epsilon < \eta/4$ or better, that is,*

$$|f(h)\cdot n - \mu| \leqslant \epsilon f(h)\cdot n < \frac{\eta}{4} f(h)\cdot n. \tag{10}$$

The following implications hold:

$$\begin{cases} \mu > m + \epsilon\cdot 2^{54} \Longrightarrow x_{10} > x_2, \\ \mu < m - \epsilon\cdot 2^{54} \Longrightarrow x_{10} < x_2, \\ |m-\mu| \leqslant \epsilon\cdot 2^{54} \Longrightarrow x_{10} = x_2. \end{cases} \tag{11}$$

Proof: First notice that $f(h) \leqslant 1$ for all h. Since $n < 2^{54}$, condition (10) implies $|\mu - f(h)\cdot n| < 2^{54}\epsilon$, and hence $|f(h)\cdot n - m| > |\mu - m| - 2^{54}\epsilon$.

Now consider each of the possible cases that appear in (11). If $|\mu - m| > 2^{54}\epsilon$, then $f(h)\cdot n - m$ and $\mu - m$ have the same sign. The first two implications from (8) then translate into the corresponding cases from (11).

If finally $|\mu - m| \leqslant 2^{54}\epsilon$, then the triangle inequality yields $|f(h)\cdot n - m| \leqslant 2^{55}\epsilon$, which implies $|f(h) - m/n| < 2^{55}\epsilon/n < 2^{53}\eta/n \leqslant \eta$. But by definition of η, this cannot happen unless $m/n = f(h)$. This accounts for the last case. ∎

Even though a 23.5 KB table for the direct table method may still be acceptable in size, let us mention a way this table could be reduced. The function $\varphi(h)$ is a staircase-function that grows slower than identity. This means several consecutive h map to a same value $g = \varphi(h)$. The corresponding values of $f(h) = 2^{\varphi(h)\cdot\log_2 5 - h}$ are binary shifts of each other. Hence, we may reduce the table size (at the price of a few additional operations) by storing the most significant bits of $f(h)$ as a function of g. We then shift the value read off the table by an appropriate amount to recover $f(h)$. The number of elements to tabulate goes down from 1504 to 647, that is, by a factor of about 2.3. This would correspond to a 10.4 KB table (for a memory alignment on 8-byte boundaries).

More precisely, define $F(g) = 5^g 2^{-\psi(g)}$ (cf. Sec. II). We then have

$$f(h) = F(\varphi(h))\cdot 2^{-\rho(h)}, \tag{12}$$

where

$$\rho(h) = h - \psi(\varphi(h)).$$

The shift $\rho(h)$ is easily computed based on Property 2. In addition, we may check that

$$1/2 < F(\varphi(h)) \leqslant 1 \quad \text{and} \quad 0 \leqslant \rho(h) \leqslant 2$$

for all h. Since $\rho(h)$ is nonnegative, multiplication by $2^{\rho(h)}$ can be implemented with a bitshift, not requiring branches. Additionally, since there is enough headroom in the 64-bit variable holding m (which fits on 54 bits), we can use a left shift on m rather than a two-word right shift on $F(g)$.

Note that we did not implement this size reduction and hence could not check its suitability in practice. Instead, we concentrated on another size reduction opportunity, that we shall describe now.

B. Bipartite table

The table size can still be reduced more, using an alternative method, based on a bipartite table. The technique takes advantage of the fact that the exact value of 5^g fits on 64 bits for $g \leqslant 27$. Assuming that the entries are 8-byte aligned, the table uses only 800 B of storage. Compared to the straightforward method from Section IV-A, this second method requires quite a few more arithmetic operations for the sake of a considerably smaller table. Most of the additional overhead in arithmetic operations can however be hidden on current processors through increased instruction-level parallelism. The reduction in table size also helps decreasing the probability of cache misses, as it holds on only a few cache lines.

Recall that we suppose $g = \varphi(h)$ and hence $-787 \leqslant h \leqslant 716$. Write

$$g = \varphi(h) = 16q - r, \qquad q = \left\lfloor \frac{g}{16} + 1 \right\rfloor, \tag{13}$$

so that $f(h) = 5^{16q}\cdot 5^{-r}\cdot 2^{-h}$. We thus have

$$-20 \leqslant q \leqslant 21, \qquad 1 \leqslant r \leqslant 16. \tag{14}$$

In particular, 5^r is an integer and fits on 38 bits.

Instead of tabulating $f(h)$ directly, we will use two tables: one containing the (exact) value 5^r for $1 \leqslant r \leqslant 16$, the other, the most significant bits of 5^{16q} to a precision of roughly 128 bits. It will prove convenient to store these values with their leading non-zero bit left-aligned, in a fashion similar to floating-point significands. They will be stored respectively on one and two 64-bit words each. Thus, we set

$$\theta_1(q) = 5^{16q}\cdot 2^{-\psi(16q)+127},$$
$$\theta_2(r) = 5^{r}\cdot 2^{-\psi(r)+63},$$

where the power-of-two factors provide for the desired left-alignment. We can check that

$$2^{127} \leqslant \theta_1(q) < 2^{128} - 1, \quad 2^{63} < \theta_2(r) < 2^{64} \tag{15}$$

for all h.

The value $f(h)$ now decomposes as

$$f(h) = \frac{\theta_1(q)}{\theta_2(r)} 2^{-64-\sigma(h)}$$

where

$$\sigma(h) = \psi(r) - \psi(16q) + h. \tag{16}$$

Equations (7) and (16) imply that

$$h - g\log_2 5 - 1 < \sigma(h) < h - g\log_2 5 + 1.$$

As we also have $h\log_5 2 - 1 \leqslant g = \varphi(h) < h\log_5 2$ by definition of φ, it follows that

$$0 \leqslant \sigma(h) \leqslant 3. \tag{17}$$

In particular, the quantity $m \cdot 2^{\sigma(h)}$ fits on 56 bits, leaving headroom for additional 8 bits on a 64-bit variable.

Now let

$$\Delta = \theta_1(q) \cdot n \cdot 2^{-64+8} - \theta_2(r) \cdot m \cdot 2^{8+\sigma(h)}.$$

Obviously, we have $|x_2| > |x_{10}|$, $|x_2| = |x_{10}|$ or $|x_2| < |x_{10}|$ depending respectively if $\Delta < 0$, $\Delta = 0$ or $\Delta > 0$. The case $|x_2| = |x_{10}|$ implies $x_2 = x_{10}$ because this second stage of the algorithm is used only when x_2 and x_{10} have the same sign.

Lemma 2. *Unless $x_2 = x_{10}$, we have $|\Delta| \geqslant 2^{124}\eta$.*

Proof: The definition of Δ rewrites as

$$\Delta = 2^{8+\sigma(h)}\theta_2(r)n\left(f(h) - \frac{m}{n}\right).$$

The bounds (5), (15), (17) together imply that $2^{8+\sigma(h)}\theta_2(r)n \geqslant 2^{124}$. We know from Theorem 1 that either $f(h) = m/n$, which is equivalent to $x_2 = x_{10}$, or $|f(h) - m/n| \geqslant \eta$. ∎

As already explained, the values of $\theta_2(r)$ are all integers and can exactly be tabulated as-is. In contrast, only an approximation can be tabulated for $\theta_1(q)$. We chose to represent these values as $\lceil\theta_1(q)\rceil$, hence replacing Δ by the easy-to-compute

$$\tilde{\Delta} = \lfloor\lceil\theta_1(q)\rceil \cdot n \cdot 2^{8-64}\rfloor - \theta_2(r) \cdot m \cdot 2^{8+\sigma(h)}. \tag{18}$$

Here n, $\theta_2(r)$ and $m \cdot 2^{8+\sigma(h)}$ are all nonnegative integers of at most 64 bits, and $\lceil\theta_1(q)\rceil$ is a positive integer of at most 128 bits.

Computing the floor function in the first term of $\tilde{\Delta}$ comes down to dropping the low-order 64-bit word in a three-word integer. This is also the reason we chose to take the ceiling $\lceil\theta_1(q)\rceil$ when representing $\theta_1(q)$ in the table: by dropping words, we can only compute under-approximations. By choosing over-approximations for the table value, we can manage for the two approximations to cancel out, at least partially. As we will see in the sequel, in particular with Theorem 3, this effect allows equality cases (cases where $x_2 = x_{10}$) to be handled extremely easily.

The multiplications by 2^8 in (18) are not essential. They serve to shift the significant bits of m and n as much as possible to the left, so that the two-word comparison that follows finishes after only one branching in as many cases as possible.

Lemma 2 is in principle enough to compare x_2 to x_{10}, using a criterion similar to that from Theorem 2. But we can actually prove finer properties that allow for a more efficient final decision step. Indeed, the use of Theorem 2 would require adding and subtracting a particular constant ϵ. Here, we show that the sign of $\tilde{\Delta}$ already provides the answer to the comparison.

Theorem 3. *Assume $g = \varphi(h)$, and let $\tilde{\Delta}$ be the 128-bit signed integer defined by* (18)*. Then following equivalences hold:*

$$\begin{aligned}
\tilde{\Delta} < 0 &\iff |x_{10}| < |x_2|,\\
\tilde{\Delta} = 0 &\iff x_{10} = x_2,\\
\tilde{\Delta} > 0 &\iff |x_{10}| > |x_2|.
\end{aligned}$$

Proof: Write

$$\lceil\theta_1(q)\rceil = \theta_1(q) + \delta_{\text{tbl}}$$

$$\lfloor\lceil\theta_1(q)\rceil \cdot n \cdot 2^{-56}\rfloor = \lceil\theta_1(q)\rceil \cdot n \cdot 2^{-56} + \delta_{\text{rnd}}$$

for some

$$0 \leqslant \delta_{\text{tbl}} < 1, \quad -1 < \delta_{\text{rnd}} \leqslant 0.$$

Setting $\delta = \delta_{\text{rnd}} + n2^{-56}\delta_{\text{tbl}}$, we have $\tilde{\Delta} = \Delta + \delta$, and, by (5),

$$-1 < \delta < 1/4.$$

If $\Delta \neq 0$, it follows that

$$|\tilde{\Delta}| \geqslant 2^{124}\eta - |\delta| > 2^{10}$$

by Lemma 2. In particular, Δ and $\tilde{\Delta}$ have the same sign, and $\tilde{\Delta} < 0 \Leftrightarrow |x_{10}| < |x_2|$. If $\Delta = 0$, we get $|\tilde{\Delta}| = \delta$, and we can conclude that $\tilde{\Delta} = 0$ because $\tilde{\Delta}$ is an integer. ∎

The conclusion $\tilde{\Delta} = 0 = \Delta$ when $x_2 = x_{10}$ means that in this case, there is no error in the approximate computation of Δ. Specifically, the tabulation error δ_{tbl} and the rounding error δ_{rnd} cancel out, thanks to our choice to tabulate the ceiling of $\theta_1(q)$.

Theorem 3 implies that the following algorithm correctly computes the sign of $x_2 - x_{10}$.

Algorithm 2. ***Step 2 (second method)****.*

- *Compute q and r as defined in* (13)*. Compute $\sigma(h)$ using Property 2.*
- *Read the 128 bits of $\lceil\theta_1(q)\rceil$ as two 64 bit words.*
- *Compute the 128 high-order bits α of*

  $$\lceil\theta_1(q)\rceil \cdot (n2^8)$$

  with one 128×64-bit multiplication, dropping the 64 low order bits.
- *Compute*

  $$\beta = \theta_2(r) \cdot (m2^{8+\sigma(h)})$$

with one full 64×64*-bit multiplication, keeping all 128 bits.*

- *Compute the signed difference* $\tilde{\Delta} = \alpha - \beta$.
- *Return*

$$\begin{cases} \text{“}x_2 > x_{10}\text{”} & \text{if } \tilde{\Delta} < 0, \\ \text{“}x_2 = x_{10}\text{”} & \text{if } \tilde{\Delta} = 0, \\ \text{“}x_2 < x_{10}\text{”} & \text{if } \tilde{\Delta} > 0. \end{cases}$$

V. Equality cases

As seen in the last Sections, the direct and bipartite methods are able to precisely determine cases when x_2 is *equal* to x_{10}, besides deciding the two possible inequalities. However, when it comes to solely determine such equality cases additional properties may simplify and accelerate the previous tests. The condition $x_2 = x_{10}$ is equivalent to

$$m \cdot 2^h = n \cdot 5^g.$$

Lemma 3. *Assume* $x_2 = x_{10}$. *Then,*

- *either* $0 \leqslant g \leqslant 22$, $0 \leqslant h \leqslant 53$, *and then* m *is divisible by* 5^g *and* n *is divisible by* 2^h*;*
- *or* $-22 \leqslant g \leqslant 0$, $-51 \leqslant h \leqslant 0$, *in which case we have* $2^{-h} \mid m$ *and* $5^{-g} \mid M_{10}$.

Proof: Notice that h and $g = \varphi(h)$ always have the same sign. When they are nonnegative, n must be a multiple of 2^h and m must be a multiple of 5^g. It follows that $5^g \leqslant m < 2^{53}$, hence $g \leqslant \log_5 2^{53} < 22.9$, which in turn implies $h \leqslant 53$. When g and h are negative, m is a multiple of 2^{-h} while $n = M_{10} \cdot 2^\nu$ and hence M_{10} are multiples of 5^{-g}. We get $-g < \log_5 10^{16} < 22.9$, whence $-h \leqslant 51$. ∎

We deduce the following algorithm, which decides whether $x_2 = x_{10}$ using only 64-bit integer operations. Note that here we do not need to compute $\varphi(h)$.

Algorithm 3. ***Equality test.***

- *if* $0 \leqslant h \leqslant 53$ *and* $0 \leqslant g \leqslant 22$ *and* $2^h \mid n$ *then*
  - *read* 5^g *in a table or compute it on the fly*
  - *compute* $m' = 5^g \cdot (n\, 2^{-h})$ *with a 64-bit integer multiplication, yielding 64 output bits*
  - *return* $(m' = m)$
- *else if* $h \geqslant -51$ *and* $-22 \leqslant g \leqslant 0$ *and* $2^{-h} \mid m$ *then*
  - *read* 5^{-g} *in a table or compute it on the fly*
  - *compute* $n' = 5^{-g} \cdot (m\, 2^h)$ *with a 64-bit integer multiplication*
  - *return* $(n' = n)$
- *else return* ***false***

The constants 5^g, $0 \leqslant g \leqslant 22$ used in the algorithm all fit on 51 bits. They can for instance be read in a small table or computed on the fly (which requires at most 5 squarings and multiplications by 5 using binary powering). The tests on g in Algorithm 3 are there only to make sure that the values of $5^{\pm g}$ used later are in the allowable range.

VI. Experimental results

Both algorithms, the direct method presented in Section IV-A and the bipartite table method presented in Section IV-B, have been implemented and thoroughly tested. Our implementation is available at

http://hal.archives-ouvertes.fr/ensl-00737881/

along with tables of worst and bad cases for the approximation problem from Section III.

To the extent possible when using decimal floating-point numbers, we gave priority to portability over performance in the main source code. It is written in plain C, using the decimal floating-point type support offered by the compiler we are using, `gcc` 4.6.3-1 [8]. No particular effort at optimization was made. For instance, multiple-word operations are implemented in portable C, with no use of hardware support for operations such as $64 \times 64 \to 128$ bit multiplication except for that automatically inserted by the compiler. Nevertheless, we also provide an alternative implementation of the bipartite table method featuring some manual optimization. This optimized version is written using `gcc` extensions and compiler intrinsics besides plain C.

Testing was done using test vectors that extensively cover all floating-point input classes, such as normal numbers, subnormals, Not-A-Numbers, zeros, infinities, for both the binary input x_2 and the decimal input x_{10}.

The test vectors also exercise the worst-cases (m, n) for the critical value $d_h(m, n) = \left|5^{\varphi(h)}/2^h - m/n\right|$ for each admissible value of h, as explained in Section III. Additionally, the 50-but-worst cases (m, n) for each value h were computed using the method described in the Appendix of [1] and added to the test vectors. The rationale behind exercising not only the worst-case for each h but also some bad cases is that an implementation might work for the worst-case input just by chance, whereas chances decrease rapidly for a larger number of difficult test cases.

The test vectors were finally completed with a reasonably large set of random inputs that fully exercise all possible values of the normalization $2^\nu M_{10}$ for the decimal mantissa (cf. Section II) as well as both the first step succeeding or the second step being necessary.

Both implemented methods have been compared for performance to the naïve comparison method where the binary or decimal input is converted to decimal or binary respectively before getting compared. These experimental results are reported in Table II. In the last two rows, “easy” cases are cases where the first step of our algorithm succeeds, while “hard” cases refer to those for which running the second step is necessary.

The code is executed on a system equipped with a quad-core Intel Core i7 M 620 processor clocked at 2.67 GHz, running Linux 3.2.0 in 64 bit mode. The comparison functions are compiled at optimization level 3, with the `-march=native` flag set. The timing measurements were done using the Read-Time-Step-Counter instruction after pipeline serialization. The

	Naïve method converting x_2 to decimal64 *(incorrect)* min/avg/max	Naïve method converting x_{10} to binary64 *(incorrect)* min/avg/max	**Direct method** described in Section IV-A min/avg/max	**Bipartite table method** described in Section IV-B min/avg/max	**Bipartite table method (optimized implementation)** min/avg/max
Special cases (± 0, NaNs, Inf)	14/–/254	7/–/281	8/–/114	7/–/132	7/–/112
x_2 subnormal, x_{10} of same sign	87/137/297	192/226/300	173/199/307	196/219/298	23/54/158
x_2 normal, x_{10} of opposite sign	50/144/278	21/107/321	5/32/118	5/30/121	4/38/122
x_2 normal, same sign, "easy" cases	84/156/294	21/107/309	**15/47/144**	**15/46/122**	**6/47/122**
x_2 normal, same sign, "hard" cases	32/95/283	56/86/207	**34/58/212**	**52/70/226**	**19/44/198**

TABLE II
TIMINGS (IN CYCLES) FOR BOTH PRESENTED METHODS AND FOR TWO NAÏVE METHODS.

serialization and function call overhead was subtracted off after timing an empty function. Measurements were taken once the caches were preheated by previous comparison function calls, the results of which were discarded. The indicated numbers are in cycles and given for the minimum/ average/ maximum value that was observed. Cycle counts larger than about 370 (depending on the test run) were discarded, and the corresponding tests run again, in order not to account in the results for long delays likely due to exceptional external events. No average values are given for case involving special inputs, such as zeros, Not-A-Numbers (NaNs) or infinities.

As can be seen from Table II, our implementations of both the Direct Method presented in Section IV-A and the Bipartite Table Method described in Section IV-B outperform both naïve methods in most cases. Let us mention again that this naïve comparison is what some C compilers currently do (cf. Section I). Besides being faster, the methods presented in this paper have the advantage of always providing correct boolean answers.

The (unoptimized) Direct Method, based on direct tabulation of $f(h)$ (cf. Section IV-A), is slightly faster than the (unoptimized) Bipartite Table Method. This is mainly due to the more computationally expensive second step of the Bipartite Table Method. The difference in performance might however be too small to justify the usage of an about 30 times larger table for the direct method. We do not have experimental data for the Direct Method using the table size reduction technique through tabulation of $F(g)$ (cf. Section IV-A).

VII. CONCLUSION

Though not foreseen in the IEEE 754-2008 standard, exact comparisons between floating-point formats of different radices would enrich the current floating-point environment and enhance the safety and provability of numerical software.

This paper has investigated the feasibility of such comparison at the instance of the binary64 and decimal64 formats. A simple test has been presented that eliminates most of the comparison inputs. For the remaining cases, two algorithms were proposed, a more direct method and a technique based on a bipartite table. The bipartite table uses only 800 bytes of table space.

Both methods have been proven, implemented and thoroughly tested. They outperform the naïve comparison technique consisting in conversion of one of the inputs to the respectively other format. Furthermore, they always return a correct answer, which is not the case of the naïve technique.

Future work will have to address the other common IEEE 754-2008 binary and decimal formats, such as binary32 and decimal128. Since the algorithmic problems of exact binary to decimal comparison and correct rounding of binary to decimal conversions (and vice versa) are similar, future investigations should also consider the possible reuse of tables for both problems. Finally, one should mention that considerable additional work is required in order to enable mixed-radix comparisons in the case when the decimal floating-point number is stored in dense-packed-decimal representation.

ACKNOWLEDGEMENT

This work is partly supported by the TaMaDi project of the French *Agence Nationale de la Recherche.*

REFERENCES

[1] M. Cornea, J. Harrison, C. Anderson, P. T. P. Tang, E. Schneider, and E. Gvozdev, *A software implementation of the IEEE 754R decimal floating-point arithmetic using the binary encoding format*, IEEE Transactions on Computers **58** (2009), no. 2, 148–162.

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

[3] IEEE Computer Society, *IEEE standard for floating-point arithmetic*, IEEE Standard 754-2008, August 2008, available at http://ieeexplore.ieee.org/servlet/opac?punumber=4610933.

[4] ISO/IEC JTC 1/SC 22/WG 14, *Extension for the programming language C to support decimal floating-point arithmetic*, Proposed Draft Technical Report, May 2008.

[5] A. Ya. Khinchin, *Continued fractions*, Dover, New York, 1997.

[6] J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres, *Handbook of floating-point arithmetic*, Birkhäuser, Boston, 2010.

[7] O. Perron, *Die Lehre von den Kettenbrüchen*, 3rd ed., Teubner, Stuttgart, 1954–57.

[8] The GNU project, *GNU compiler collection*, 1987–2013, available at http://gcc.gnu.org/.