New Results on the Distance Between a Segment and $\mathbb{Z}^2$. Application to the Exact Rounding

Vincent Lefèvre

LORIA / INRIA Lorraine
615 rue du Jardin Botanique
54602 Villers-lès-Nancy Cedex, France
E-mail: vincent@vinc17.org

Abstract

This paper presents extensions to Lefèvre's algorithm that computes a lower bound on the distance between a segment and a regular grid $\mathbb{Z}^2$. This algorithm and, in particular, the extensions are useful in the search for worst cases for the exact rounding of unary elementary functions or base-conversion functions. The proof that is presented here is simpler and less technical than the original proof. This paper also gives benchmark results with various optimization parameters, explanations of these results, and an application to base conversion.

1. Introduction

The problem of computing a lower bound on the distance between a segment and $\mathbb{Z}^2$ was introduced in [4] to search for worst cases for the exact rounding of unary elementary functions (exp, log, sin, cos...), as explained in [7, 5, 6]. Though this algorithm can be used for other problems, the context is briefly recalled here, as the input data for this problem are not completely random and sometimes are very particular, so that they must be taken into account to improve the algorithm.

In general, the result of an operation on floating-point numbers is not exactly representable in the target floating-point system (either the same one or a different one, such as in base conversions): it must be *rounded* to a representable number, called *machine number*. The IEEE-754 standard ([3, 1]) defines four rounding modes: towards $-\infty$, $+\infty$, 0 (called *directed* rounding modes) and to nearest. This standard requires that the results of the arithmetic operations $+$, $-$, $\times$, $\div$, and the square root be *exactly* (or *correctly*) rounded, as if the results were first computed with "infinite precision", then rounded accordingly to the chosen rounding mode. Providing exact rounding for the above operations is very easy and well supported nowadays. We wish to extend this requirement to the elementary functions (exponential, logarithm, trigonometric and hyperbolic functions, etc.) too, in IEEE-754 double precision (i.e., with a 53-bit mantissa), and possibly in some higher precisions; this is much more difficult, and this explains why there are no such requirements yet. This paper will also deal with base conversion (in Section 5), but first, let us focus on unary elementary functions. Only information necessary to understand the problem is given here; the reader can find more details about the context in [7, 5, 6].

Let us consider a floating-point system and a unary elementary function f. Currently, the only solution to provide exact rounding at a guaranteed reasonable cost is to perform an exhaustive search for the *worst cases*, that is, the cases that are the hardest to round exactly. A worst case corresponds to a machine number x such that $f(x)$ is very close to a machine number (for the three directed rounding modes) or to the middle of two consecutive machine numbers (for the rounding to nearest mode). As these numbers are spaced in a regular way (except when the exponent changes, but this is rare and it is easy to cope with that), the problem can be rephrased: *Find the points of a regular grid that are "close enough" to the graph of the function.* Note that due to the symmetry of the problem, the inverse function f^{-1} can generally be tested at the same time.

The domain on which the function is to be tested is split into small subdomains where the function can be approximated by a degree-1 polynomial, i.e., the graph of the function is approximated by segments. Then the algorithm described in this paper can be used on each subdomain to find the points of the regular grid (which can be seen as $\mathbb{Z}^2$ after scaling) that are the closest to the segment. For instance, if we consider the double precision, one fixed exponent (e.g., $1 \leq x < 2$), and all the possible 54-bit mantissas (a bit has been added to the 53 bits of the floating-point system in order to be able to test the inverse function f^{-1}), the segment of each subdomain will contain several thousands of points whose abscissa is an integer and there will be several billions of subdomains. As we are interested in no more than a few thousands of worst cases, a very large proportion of these subdomains will have no points close enough to the segment. Therefore we are also interested in the following problem, simpler than the previous one: "*Are there points of a regular grid that are close enough to a segment?*" where the answer can be "no", "yes" or "maybe". In general, the answer will be "no" (regarded as *success*). In the few cases where it is "yes" or "maybe" (regarded as *failure*), we can go back to the previous problem and find these points. [4] dealt with this simpler problem only and a slow, naive algorithm was used in the few cases of failure.

Instead of degree-1 polynomials, degree-2 polynomials can be used to approximate the function and reach a better asymptotic complexity, as described in [8]. However, the algorithm in [8] is much more complex. For the double precision, it is preferable to choose degree-1 polynomials. For the extended precision (64-bit mantissa), the best choice depends very much on improvements in the algorithms and on the implementation.

Here, we look for an efficient algorithm and implementation for our problem. For a more theoretical point of view and a more general case, the reader may look at [2].

First we describe the new algorithm and give a simpler proof than the one given in [4] (Section 2). It is still an extension of the subtractive Euclidean algorithm to compute a GCD. Using divisions instead of subtractions leads to a better complexity; so, in Section 3, we discuss how to replace subtractions by divisions in an efficient way. Then, in Section 4, we give benchmarks with various parameters (on the one hand, variants of the algorithms, and on the other hand, inputs with different properties), compare the obtained results and explain them. In Section 5, we give an application to the search for worst cases for the base conversion. Section 6 is the conclusion, where we briefly discuss on future improvements.

2. A New Algorithm

We consider a segment $y = b - ax$, where a and b are real numbers and x is restricted to a given interval: $0 \leq x < N$, N being a positive integer. We are interested in the values $k \in [\![0, N-1]\!]$ (the integers k satisfying $0 \leq k \leq N-1$) such that $\{b - k.a\} < d_0$, where $\{y\}$ denotes the positive fractional part of y, and d_0 is a positive real number (less than 1).

Throughout this section, the reader can look at a very simple example in Figure 1.

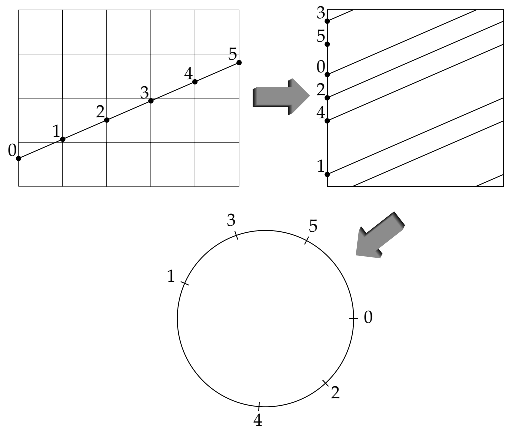

Figure 1. The segment and integer grid, the 2-dimensional transformation modulo 1, and the representation of the left segment (corresponding to $x \in \mathbb{Z}$) modulo 1 as a circle.

2.1. Mathematical Preliminaries

In the following, we will work in $\mathbb{R}/\mathbb{Z}$, the additive group of the real numbers modulo 1. This set can be viewed as a circle, or as a segment $[0, 1]$ where the boundaries 0 and 1 are identified. With this second representation, the point represented by 0 can be regarded as an origin. The operation consisting in adding an element $\alpha \in \mathbb{R}/\mathbb{Z}$ can be regarded as a translation on the segment (or a rotation on the circle).

If $a \in \mathbb{R}/\mathbb{Z}$ and k is a nonnegative integer, k is said to be the (group) *index* of $k.a$ (in the additive group generated by a).

If $y \in \mathbb{R}$, its image in $\mathbb{R}/\mathbb{Z}$ is also denoted y, as there is no possible confusion, and the real $\{y\}$ is regarded as the canonical representative of y in $\mathbb{R}/\mathbb{Z}$.

2.2. Properties of $k.a$ modulo 1

If a is a rational number, then there exists a positive integer k such that $k.a = 0$ in $\mathbb{R}/\mathbb{Z}$. For the mathematical study, we assume that a is irrational to avoid such casual equalities, even though the algorithm will be applied on rational numbers (one may assume that a is a rational plus an arbitrary small irrational number, or that a is rational but N is small enough so that such casual equalities do not occur).

In the following, we study the configurations $\{k.a \in \mathbb{R}/\mathbb{Z} : k \in \mathbb{N},\ k < n\}$ for $n \geq 2$. But before going into the details, let us explain how the configurations evolve when n increases.

In any configuration, the n points $k.a$ split the segment $[0, 1]$ into n intervals, and one can notice that there are some particular values of n such that these intervals have exactly two possible values h and ℓ ($h > \ell$). One can also notice that when points are added to such a configuration (by increasing n), each new point splits some interval of length h into an interval of length ℓ and an interval of length $h - \ell$. Once all the intervals of length h have been split, one obtains a particular configuration with intervals of lengths ℓ and $h - \ell$ only. Note that replacing $\{h, \ell\}$ by $\{\ell, h - \ell\}$ corresponds to a step of the subtractive Euclidean algorithm to compute a GCD; this similarity will be used in Section 3.

The fact that the intervals can have at most three possible lengths (h, ℓ and $h - \ell$) in any configuration is known as the *three-distance theorem*: on a circle with points added by successive constant rotations, the distances between two adjacent points of any configuration can take at most three possible values. The circle in Figure 1 gives such an example.

The configurations $\{k.a \in \mathbb{R}/\mathbb{Z} : k \in \mathbb{N},\ k < n\}$ having only two interval lengths are studied by induction on the number of points n, as shown on Figure 2. We seek to prove that each configuration has the following properties (the values of n, u, v, x, $y\ldots$ depend on the configuration).

- The n points $0.a$ (the origin), $1.a$, $2.a$, …, $(n-1).a$ modulo 1 split the segment $[0, 1]$ into u intervals of length x and v intervals of length y, where $n = u + v$.
- The intervals of length x are denoted x_0, x_1, …, x_{u-1}, where x_0 is the left-most interval of $[0, 1]$ and $x_r = x_0 + r.a$ (i.e., x_r is x_0 translated by $r.a$ in $\mathbb{R}/\mathbb{Z}$).
- The intervals of length y are denoted y_0, y_1, …, y_{v-1}, where y_0 is the right-most interval of $[0, 1]$ and $y_r = y_0 + r.a$ (i.e., y_r is y_0 translated by $r.a$ in $\mathbb{R}/\mathbb{Z}$).

In the initial configuration, $n = 2$ and $u = v = 1$. The left interval x_0 has the length $x = \{a\}$ and the right interval y_0 has the length $y = 1 - \{a\}$.

Now, let us explain how one goes from a configuration with n points (where the above properties are considered to be satisfied) to the next one and why the properties are still satisfied in the next configuration.

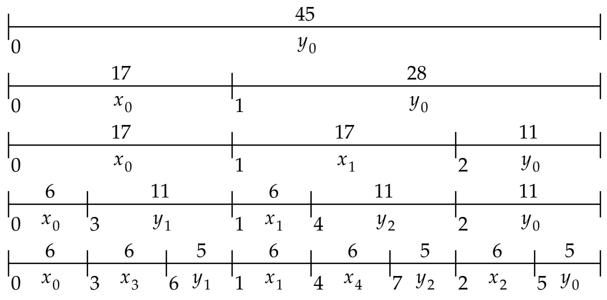

Figure 2. Configurations for $a = 17/45$. The points $k.a$ for $0 \leq k < n$ are represented by the vertical bars, with the values of k next to them. The name of each interval (x_r or y_r) and the corresponding length (scaled by 45) are given. The initial configuration considered in this section has two points and two intervals, but a virtual configuration with one point (0) and one interval is shown here for completeness, as it may be used in some codes.

First, let us add the next point; it is $n.a$. We assumed above that all the points $k.a$ were different in our domain. Therefore $n.a$ is between two points whose indices are less than n. Amongst these two points, at least one of these indices, denoted r, is nonzero.

In the configuration with n points, the points of indices $r-1$ and $n-1$ exist (since $r \geq 1$) and are adjacent: They are obtained by a translation by $-a$. Their distance is either x or y (the only two possible distances between adjacent points), and the distance between the points of indices r and n (after $n.a$ has been added) is the same one since these points are obtained by a translation. Thus the point of index n splits an interval of length $h = \max(x, y)$ into two intervals of respective lengths $\ell = \min(x, y)$ and $h - \ell$. The length $h - \ell$ is new[1]; therefore the corresponding interval does not have an inverse image (i.e., an image by the translation by $-a$), and one of its boundaries is the point of index 0 (the only point that does not have an inverse image).

The following steps consist of successive translations by a. By induction, this means that each interval x_1, x_2, …, x_{u-1} (if $h = x$) or y_1, y_2, …, y_{v-1} (if $h = y$) are split the one after the other in the same way as above, and we consider the next configuration where all the intervals of length h have been split into two intervals of respective lengths ℓ and $h - \ell$.

[1]The case $h - \ell = \ell$ has been rejected as it would lead to $k.a = 0 \pmod 1$.

Now, let us prove that the new configuration satisfies the properties. Without loss of generality and to make the following notations clearer, we assume that $h = x$ and $\ell = y$ (in the opposite case, exchange x and y, u and $v\dots$). So, in the new configuration, we have $u' = u$ intervals of length $x' = h - \ell = x - y$ and $v' = u + v$ intervals of length $y' = \ell = y$ (v intervals from the previous configuration and u intervals from the splits). This proves the first property.

For $0 \le r < u$, we define x'_r as the interval of length x' obtained by splitting the interval x_r. It has been shown above that the first interval of length $h - \ell$ was obtained by splitting x_0 or y_0 (since one of its boundaries was 0); here $h = x$, therefore x_0 was the first split interval, and x'_0 is the new left-most interval (still because it has been shown above that it has 0 as a boundary, and it cannot be the right-most interval since it is still y_0). Since $x_r = x_0 + r.a$, we have $x'_r = x'_0 + r.a$. This proves the second property.

Concerning the third property, we first define $y'_r = y_r$ for $0 \le r < v$. In particular, y'_0 is still the right-most interval in the new configuration. For $0 \le r < u$, we define y'_{v+r} as the interval of length y' (i.e., ℓ) obtained by splitting the interval x_r. The property $y'_r = y'_{r-1} + a$ is trivial except for $r = v$. As it has been said, y'_v is the image of an interval y_k by the translation by a; it cannot be y_j for $0 \le j \le v - 2$, as its image is y_{j+1}. Therefore it is y_{v-1}, and since $y'_{v-1} = y_{v-1}$, we have: $y'_v = y'_{v-1} + a$. This proves the third property. □

We also have the following properties:

- The lower boundary of an interval x_r has index r. This is a direct consequence of the second property.
- The lower boundary of an interval y_r has index $u + r$. This is a direct consequence of the third property, using the fact that the set of these indices for the intervals of length y must be $[\![u, u + v - 1]\!]$ since the indices of $[\![0, u - 1]\!]$ are already for the intervals x_r.

2.3. The Algorithm

Returning *all* the integers k satisfying $\{b - k.a\} < d_0$ would be tricky if we want to take into account all the possible cases, and the obtained algorithm would not be very efficient in our particular context, where there are very few such integers (and in general, none). The analysis in Section 2.2 allows us to get the first such integer (if any) very easily, as shown below. So, the algorithm presented in this section returns the first integer $r \in [\![0, N - 1]\!]$ such that $\{b - r.a\} < d_0$ if there is one, else a value larger or equal to N. If $r < N - 1$, to find the other integers, one can just subtract $(r + 1).a$ from b and $r + 1$ from N, and rerun the algorithm. In the case the segment approximates a curve, one can also add some correction terms to a and/or b to take into account a part of the approximation error.

The algorithm is directly based on the properties given in Section 2.2. Though these properties deal with many variables, only the following data are useful to find the requested result: the interval lengths x and y; the respective numbers of these intervals u and v; a binary value saying whether b is in an interval[2] of length x or in an interval of length y (in fact, it is better to have two separate codes for these two cases, instead of representing this binary value by a variable); the index r of this interval containing b; the distance d between b and the lower boundary of this interval.

Algorithm 1 — Returns the first value $r \in [\![0, N - 1]\!]$ such that $\{b - r.a\} < d_0$ if there is such an integer, else an integer larger or equal to N.

```
*Initialization:*  $x = \{a\}$ ; $y = 1 - \{a\}$ ; $d = \{b\}$ ;
                   $u = v = 1$ ; $r = 0$ ;
**if** $(d < d_0)$ **return** 0      [for any bracket: $ux + vy = 1$]
*Unconditional loop:*
  **if** $(d < x)$
    **while** $(x < y)$
      **if** $(u + v \ge N)$ **return** $N$      $[h = y,\ b \in x_r]$
      $y = y - x$;  $u = u + v$;
    **if** $(u + v \ge N)$ **return** $N$      $[h = x,\ b \in x_r]$
    $x = x - y$;
    **if** $(d \ge x)$ $r = r + v$;
    $v = v + u$;
  **else**
    $d = d - x$;
    **if** $(d < d_0)$ **return** $r + u$      $[b \in y_r]$
    **while** $(y < x)$
      **if** $(u + v \ge N)$ **return** $N$      $[h = x,\ b \in y_r]$
      $x = x - y$;  $v = v + u$;
    **if** $(u + v \ge N)$ **return** $N$      $[h = y,\ b \in y_r]$
    $y = y - x$;
    **if** $(d < x)$ $r = r + u$;
    $u = u + v$;
```

This gives Algorithm 1. This is just one form of the algorithm. There are different variants; for instance, some loops could be unrolled and/or some tests could be performed at different places. Note: the test $d < x$ at the beginning of the unconditional loop is performed

[2]In order to match the definition of the positive fractional part, the intervals are closed at the lower boundary and open at the upper boundary, i.e., only the lower boundary belongs to the intervals.

when the interval containing b is split, and it decides whether b goes to the new interval of length x (true) or the new interval of length y (false); hence the separate codes mentioned above.

2.4. Discussion on this Algorithm

In Section 2.2, we supposed that one could not get casual equalities. As such equalities may occur in practice, we need to talk about them. In fact, the real problem is when a length (either x or y) becomes null. This can happen only if r has not been found yet (i.e., $d \geq d_0$ in the current configuration). In the mathematical point of view, all the subsequent points will be points that have already been found; this means that for any $k \in \mathbb{N}$, there are no points of index k such that $\{b - k.a\} < d_0$. Therefore we must return a value r larger or equal to N, and this is exactly what the algorithm does. One could add tests $x = 0$ and $y = 0$, but this is not necessary here and would slow down the algorithm (the case $x = 0$ or $y = 0$ is very rare in practice).

Now, let us talk about the types of the variables. The algorithm uses two kinds of variables: variables representing cardinals or indices (u, v, r) and variables representing lengths (x, y, d, d_0), without taking into account a and b (they disappear at the very beginning, and implementations can even avoid them completely). For the former kind of variables, integer arithmetic is naturally used; $u + v$ and r are bounded above by $2N$, so that integer overflows can easily be avoided. For the latter kind of variables, fixed-point arithmetic (possibly modulo 1) is naturally used; this can be done either with conventional unsigned (modular) integer arithmetic or with floating-point arithmetic, in which case, the inputs should properly be rounded to ulp(1/2) so that the operations are performed exactly.

We can also notice that a large number of consecutive subtractions (and associated additions) may be performed before the sign of $x - y$ changes, so that a division would be faster. This is the topic of the next section.

3. Replacing Subtractions by Divisions

As noted in Section 2.2, Algorithm 1 is very similar to the Euclidean algorithm to compute a GCD: reducing Algorithm 1 to variables x and y consists of the steps of the subtractive Euclidean algorithm. The complexity of this subtractive algorithm on two numbers less than n is $O(\log(n)^2)$ in average and $O(n)$ in the worst case. But the variance is very high and the average complexity does not have much sense, in particular in our context, where the inputs are not completely random and where we do not compute a GCD (the halting conditions are different). So, though the conventional Euclidean algorithm (i.e., with divisions) has a better average complexity, $O(\log(n))$, a modified algorithm to use divisions instead of subtractions would not necessarily be faster in practice: the probability of getting a large quotient is very low and a division is *much slower* than a subtraction. However, since very bad cases occur in practice (see examples in Section 4), using divisions would be interesting to avoid particular instances taking a very long time (this is more a problem than the average running time).

Let us denote s and t the numbers involved in the division. To avoid a division s/t when the quotient is low, we can compare s shifted a few bits to the right (denoted $s \gg c$, for some small constant positive integer c) and t, since shift operations are very fast, and decide whether a division or a sequence of subtractions should be performed. We have chosen to shift s to the right instead of shifting t to the left in order to avoid a possible overflow. Moreover, if a multiple-precision arithmetic is used, it is wiser to use approximations to s and t for this comparison so that it is faster.

Replacing subtractions by divisions is more complicated than in the conventional Euclidean algorithm since we have tests on other variables. Again, one can obtain several variants. Algorithm 2 is one of them. If floating-point arithmetic is used for the lengths, then the active rounding mode should be set to rounding towards minus infinity or towards zero to be sure that the floor on the mathematical result (corresponding to an integer division) is obtained.

Tests $x = 0$ and $y = 0$ have been added to avoid divisions by zero. Also, more care must be taken concerning possible integer overflows on q, u and v. The test $q \geq N$ is a way to avoid some of them (it may be unneeded). This really depends on the implementation and on the context, but one has the following property. If 1, x and y are integer multiples of some ε, then, as long as $x \neq 0$ and $y \neq 0$, we have: $u < \varepsilon^{-1}$, $v < \varepsilon^{-1}$ and $q \leq \varepsilon^{-1}$; and if $x = 0$ or $y = 0$, then u or v reaches ε^{-1}.

Moreover, a test $c = 0$ has been added as a special case to eliminate the comparison and force the division (of course, c is a value known at compile time).

4. The Influence of the Parameters

The algorithms have been implemented in ISO C99 for the exhaustive tests of the elementary functions,

Algorithm 2 — Still returns the first integer $r \in [\![0, N-1]\!]$ such that $\{b - r.a\} < d_0$ if there is one, else a value larger or equal to N, but this algorithm uses divisions.

```
*Initialization:*   $x = \{a\}$ ; $y = 1 - \{a\}$ ; $d = \{b\}$ ; $u = v = 1$ ; $r = 0$ ;
**if** $(d < d_0)$ **return** $0$
*Unconditional loop:*
  **if** $(d < x)$
      **if** $(((x - d) \gg c) \geq y)$
          $q = \lfloor (x-d)/y \rfloor - 1$;
          **if** $(q \geq N)$ **return** $N$
          $x = x - q \times y$;  $v = v + q \times u$;
      **if** $(c = 0$ **or** $(y \gg c) > x)$
          $q = \lfloor y/x \rfloor$ ;
          **if** $(q \geq N)$ **return** $N$
          $y = y - q \times x$;
          **if** $(y = 0)$ **return** $N$
          $u = u + q \times v$;
      **else**
          **while** $(x < y)$
              **if** $(u + v \geq N)$ **return** $N$
              $y = y - x$;  $u = u + v$;
      **if** $(u + v \geq N)$ **return** $N$
      $x = x - y$;
      **if** $(x = 0)$ **return** $N$
      **if** $(d \geq x)$ $r = r + v$;
      $v = v + u$;
  **else**
      $d = d - x$;
      **if** $(d < d_0)$ **return** $r + u$
      **if** $(c = 0$ **or** $(x \gg c) > y)$
          $q = \lfloor x/y \rfloor$ ;
          **if** $(q \geq N)$ **return** $N$
          $x = x - q \times y$;
          **if** $(x = 0)$ **return** $N$
          $v = v + q \times u$;
      **else**
          **while** $(y < x)$
              **if** $(u + v \geq N)$ **return** $N$
              $x = x - y$;  $v = v + u$;
      **if** $(u + v \geq N)$ **return** $N$
      $y = y - x$;
      **if** $(y = 0)$ **return** $N$
      **if** $(d < x)$ $r = r + u$;
      $u = u + v$;
```

using mixed 32-bit and 64-bit integer arithmetic[3]. Different parameters, described below, can be chosen, and the resulting timings are given, compared and explained in this section.

The first parameter is the shift count c used to decide whether a division or a sequence of subtractions should be performed. In the following timings, 6 possibilities are tested: the algorithms with the divisions for $c = 0$, 1, 2, 3 and 5, and the subtractive algorithms.

Another parameter is the way the algorithms are used. Results for up to 4 possibilities are given. Here is a short explanation for each of them:

1. "–": The algorithm described in [4], possibly with divisions (depending on the first parameter), is used. It is identical to the algorithms described here without the computations related to r and where the final value of d gives a lower bound on $\{b - k.a\}$; if this lower bound is "high enough" (success), one immediately deduces that there are no worst cases in the domain, otherwise (failure) the $O(n)$ naive algorithm is used to search for possible worst cases.

2. "`l=3`": The same algorithm is used, but if it fails, the interval is split into $2^3 = 8$ subintervals and this algorithm is used on each of these subintervals (if it fails on a subinterval, then the naive algorithm is used on this subinterval).

3. "`w`": The same algorithm is used first, but if it fails, then the algorithms described in this paper are used (instead of the naive algorithm).

4. "old `w`" (in Table 4): The algorithms described in this paper are used immediately.

Each test was performed 3 times on a 2 GHz AMD Opteron machine at MEDICIS[4], and the median time was kept. We checked that the output did not depend on the parameters[5].

These tests were performed for the two functions $\exp x$ and $\sin x$ in domains starting at $x = 1$ (exponent 0) and $x = 2^{-6}$ (exponent -6), for $\exp x$ in a domain starting at $x = 2^2 = 4$ (exponent 2), and for $\exp x$ in two domains around $x = \log(4)$. In each of the following tables, timings within 10% of the best timing

[3]This choice is not the best one, but this was just a patch to the tests described in [5].

[4]`http://www.medicis.polytechnique.fr/`

[5]These tests were also run on a cluster of machines and this check revealed a bug in the OS (OpenMosix).

are underlined, and timings which took at least twice as long are in boldface.

Table 1 gives timings for $\exp x$ and $\sin x$ in domains starting at $x = 1$. This corresponds to the general case. Concerning the influence of the parameter c, $c = 0$ is the slowest one, and other tests (not all included in this paper) show that it is always slower than with small positive values of c. The subtractive algorithms give the best timings (because the corresponding quotients are small, at least the few first ones); otherwise $c = 3$ is globally the best choice, not very far behind the subtractive one. Concerning the second parameter (– / `l=3` / `w`), the first choice is the slowest due to the naive method. The use of this naive method is significantly reduced with the second choice and eliminated with the third choice.

Table 1. Fct $\exp x$ and $\sin x$, exponent 0.

	$\exp x$			$\sin x$		
`c`	–	`l=3`	`w`	–	`l=3`	`w`
0	**42.30**	**35.46**	**35.26**	**40.24**	**31.72**	**31.67**
1	26.32	19.27	19.09	28.28	19.52	19.49
2	24.29	17.09	17.04	26.62	17.79	17.78
3	24.09	16.82	16.85	26.41	17.54	17.55
5	24.47	17.29	17.29	27.15	18.36	18.32
–	21.54	14.23	14.26	23.71	14.74	14.85

Table 2 gives timings for $\exp x$ and $\sin x$ in domains starting at $x = 2^{-6}$, i.e., where is x relatively small, so that the value of $\{a\}$ is particular, with large partial quotients early in its continued-fraction expansion ($\exp x$: 63, then 128; $\sin x$: 1, then 4095). This explains why the subtractive algorithm is much slower than in the general case, in particular for $\sin x$. This is even worse for smaller exponents.

Table 2. Fct $\exp x$ and $\sin x$, exponent -6.

	$\exp x$			$\sin x$		
`c`	–	`l=3`	`w`	–	`l=3`	`w`
0	18.29	18.15	18.09	15.74	15.56	15.59
1	12.54	12.52	12.51	10.22	10.06	10.10
2	12.25	12.10	12.00	9.67	9.55	9.57
3	12.10	11.95	11.86	9.45	9.25	9.26
5	14.41	14.31	14.16	9.34	9.16	9.20
–	22.13	21.94	21.97	**314.8**	**314.3**	**314.6**

Table 3 gives timings for $\exp x$ in a domain (4 times as small as in the previous tables) starting at $x = 2^2 = 4$. The error term for the approximation to $\exp x$ by a degree-1 polynomial is much larger than with smaller exponents. As the error term must be added to d_0, d_0 is much larger and the algorithm used in the first choice ("–") fails much more often. The split into 8 subintervals ("`l=3`") brings back failures to an acceptable rate. With the third choice ("`w`"), the algorithms presented in this paper often stop before the end of the interval, finding a value r that does not correspond to a potential worst case, due to the large error term, but they are still very fast.

Table 3. Function $\exp x$, exponent 2.

`c`	–	`l=3`	`w`
0	**43.55**	**11.39**	**9.63**
1	**40.00**	6.36	5.43
2	**39.46**	5.57	4.82
3	**39.37**	5.40	4.73
5	**39.47**	5.61	4.86
–	**38.82**	4.56	4.11

Table 4 gives timings for $\exp x$ in two domains around $x = \log(4)$. This is a particular case because the derivative ($\exp x$) is very close to a simple rational (4) and this means that a will have large partial quotients early in its continued-fraction expansion. This explains why the subtractive algorithms are very slow. The results concerning "`w`" and "old `w`" in the right part of the table ($x > \log(4)$) are explained by the fact that in Algorithm 2, divisions are not performed when the value of d (or r) would be modified, making it similar to subtractive algorithms; this could be a future improvement.

5. The Application to the Exactly-Rounded Base Conversion

Algorithm 2 can also be used to find worst cases for base conversion of floating-point numbers. The conventional method based on continued fractions can also be used (in fact, Algorithm 2 is an extension of this method). But here, we can restrict the input to any interval; this may particularly be useful to find difficult cases in very high precision, that could be used as tests for libraries with exact rounding like MPFR[6].

For instance, let us consider the conversion from base 2 to base 10, in the rounding to nearest mode, with n digits in base 2 and N digits in base 10. Any positive number has a binary representation of the form $f \times 2^{e-n}$, where the binary exponent e is an integer and $2^{n-1} \leq f < 2^n$, and a decimal representation of the form $F \times 2^{E-N}$, where the decimal exponent E is an integer and $2^{N-1} \leq F < 2^N$; before the rounding, F is not necessarily an integer. We have $E = \lfloor \log_{10}(f \times 2^{e-n}) + 1 \rfloor$.

If n, N and e are fixed, then E can take at most two values. So, we can split the interval and regard E as fixed too. Then, let us define k by $f = 2^{n-1} + k$, so that $F = b - k.a$ with $a = -10^{N-E} \times 2^{e-n}$ and $b = 10^{N-E} \times 2^{e-1}$.

[6]`http://www.mpfr.org/`

Table 4. Function $\exp x$, $x \approx_< \log(4)$ (domain 50624) and $x \approx_> \log(4)$ (domain 50632).

	domain 50624				domain 50632			
`c`	–	`l=3`	`w`	old `w`	–	`l=3`	`w`	old `w`
0	**1.67**	1.06	1.03	1.03	**1.15**	0.59	**7.72**	**1653**
1	1.37	0.77	0.77	$\underline{0.73}$	**1.09**	$\underline{0.56}$	**1.75**	**279**
2	1.36	0.76	$\underline{0.73}$	$\underline{0.74}$	**1.11**	$\underline{0.57}$	**1.66**	**259**
3	1.35	$\underline{0.72}$	$\underline{0.72}$	$\underline{0.70}$	**1.10**	$\underline{0.58}$	**1.69**	**259**
5	1.35	$\underline{0.73}$	$\underline{0.70}$	$\underline{0.68}$	1.04	$\underline{0.55}$	**1.68**	**259**
–	**40.42**	**40.54**	**40.19**	**48.72**	**230**	**230**	**230**	**323**

Worst cases correspond to values of F that are very close to the middle of two consecutive integers. If we want to search for worst cases that are at most at a distance ε of the middle of two consecutive integers, we just need to apply our algorithm after adding $1/2+\varepsilon$ to b and taking $d_0 = 2\varepsilon$.

An implementation was performed using the MPFR library to do some computations in interval arithmetic and obtain E, a and b. Algorithm 2 was implemented using the `mpn` layer of the GMP library[7].

Here is the worst case that has been found after a few days, in a search with $n = 53$, $N = 17$ and 18, and $|e| \leq 10^9$:

$$\begin{aligned} x &= 8296938838833989 \times 2^{377450238-53} \\ &= \underbrace{29705494656714363}_{\text{17 digits}}.5\underbrace{000\ldots000}_{\text{24 zeros}}146\ldots \\ &\quad \times 10^{113623844-17}. \end{aligned}$$

6. Conclusion

This paper presented several improvements on the algorithm introduced in [4]. There are probably other ways of doing more optimizations, in particular in our context: the search for worst cases for the exact rounding of elementary functions (or base conversion).

- The first bits of a and b could be read at the beginning to execute special code that would perform the first few steps more quickly. However, one would probably not gain very much.

- In general, the value of a does not change very much from one interval to the next one. This means that the first steps on consecutive intervals are often similar. It would be interesting to share some parts of the computations, if possible.

- In the context of the search for worst cases for elementary functions, there is an error term (which could be approximated by a degree-2 term). If the approximation by the degree-1 polynomial is performed so that the degree-2 term is 0 at the beginning of the interval, then a non-constant error term could be taken into account in d_0 when one goes farther in the domain (i.e., when $n = u + v$ increases because new points are considered), instead of adding the maximum error term for a fixed interval.

[7]`http://www.swox.com/gmp/`

References

[1] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, Mar. 1991. An edited reprint is available at `http://cch.loria.fr/documentation/IEEE754/ACM/goldbergSUN.ps` from Sun's Numerical Computation Guide; it contains an addendum *Differences Among IEEE 754 Implementations*.

[2] D. S. Hirschberg and C. K. Wong. A polynomial-time algorithm for the knapsack problem with two variables. *Journal of the ACM*, 23(1):147–154, Jan. 1976.

[3] IEEE standard for binary floating-point arithmetic. ANSI-IEEE Standard 754-1985, New York, 1985. Approved March 21, 1985: IEEE Standards Board, approved July 26, 1985: American National Standards Institute, 18 pages.

[4] V. Lefèvre. An algorithm that computes a lower bound on the distance between a segment and $\mathbb{Z}^2$. In *Developments in Reliable Computing*, pages 203–212. Kluwer, Dordrecht, Netherlands, 1999.

[5] V. Lefèvre. *Moyens arithmétiques pour un calcul fiable.* PhD thesis, École Normale Supérieure de Lyon, Lyon, France, Jan. 2000.

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

[7] V. Lefèvre, J.-M. Muller, and A. Tisserand. Towards correctly rounded transcendentals. *IEEE Transactions on Computers*, 47(11):1235–1243, Nov. 1998.

[8] D. Stehlé, V. Lefèvre, and P. Zimmermann. Worst cases and lattice reduction. In J.-C. Bajard and M. 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.