MPDI: A Decimal Multiple-Precision Interval Arithmetic Library*

S. Graillat
Sorbonne Universités, UPMC Univ Paris 06, CNRS,
UMR 7606, LIP6, F-75005 Paris, France
stef.graillat@upmc.fr

C. Jeangoudoux
Sorbonne Universités, UPMC Univ Paris 06, CNRS,
UMR 7606, LIP6, F-75005 Paris, France
Safran Electronics & Defense
clothilde.jeangoudoux@lip6.fr

Ch. Lauter
Sorbonne Universités, UPMC Univ Paris 06, CNRS,
UMR 7606, LIP6, F-75005 Paris, France
christoph.lauter@lip6.fr

Abstract

We describe the mechanisms and implementation of a library that defines a decimal multiple-precision interval arithmetic. The aim of such a library is to provide guaranteed and accurate results in decimal arithmetic. This library contains correctly rounded (for decimal arbitrary precision arithmetic), fast and reliable basic operations and some elementary functions. Furthermore the decimal representation is IEEE 754-2008 compatible, and the interval arithmetic is compliant with the new IEEE 1788-2015 *Standard for Interval Arithmetic* [2].

Keywords: interval arithmetic, multi-precision, IEEE 754-2008, decimal arithmetic.
AMS subject classifications: 65-G30

1 Introduction

In the aeronautical industry, embedded software is designed to answer a set of requirements describing its behaviour. Those requirements manipulate decimal numbers, hence describe a decimal system. To implement the behaviour described in the requirements, the software performs binary computation. The conversion between decimal and binary introduces a number of well-known issues: a finite decimal number is not always representable in binary, and the error introduced in this conversion is often

*Submitted: December 9, 2016; Revised: June 27, 2017; Accepted: June 29, 2017.

amplified along with the combination of operations. The same issues of conversion between binary and decimal numbers also appear in financial applications.

First we can wonder why it should be reasonable to use decimal instead of binary to begin with. Indeed, the easiest way to avoid such conversion errors is simply to stop designing systems intended to be run on a computer in decimal arithmetic and do it all in binary. But in many cases this is not possible, due to the difference of cultural background and history of the field, as in the aeronautical industry, or for some financial applications.

The tricky part comes when trying to verify the behaviour of the software according to the requirements. On one hand we have a decimal system that the software is supposed to realise and on the other hand we have the binary results of the software. To be able to guarantee the conformance of the binary software with the decimal system, we might want to implement this system using decimal arithmetic. In the case of financial applications, manipulating decimal arithmetic to represent money prevents the introduction of conversion errors. That is why there is a need for robust and correctly rounded decimal arithmetic.

Second we might want to know why it is interesting to use interval arithmetic. It is very common that applications developed in those fields have critical features. Interval arithmetic consists in representing a number by a certified enclosure instead of an approximated floating-point number. Hence every operation is done with intervals enclosing the real numbers, and computes a result enclosed in an interval too. In the case of critical applications, interval arithmetic can be very useful during the validation and testing process, to ensure a bound on the software error.

And third we might question why we want to use multiple precision with interval arithmetic. Multiple-precision enables the user to choose the precision of each variable. This precision can be arbitrarily large, and allows us to compute an operation with more digits than in double precision for example. In order to compute the bounds of an interval, we use directed rounding modes. Using multiple-precision arithmetic can in most cases minimize the error made by the computation on those bounds and so the resulting intervals are as narrow as possible.

Combining all those motivations, we can see that there is a need for a decimal multiple-precision interval library.

A lot of work has been done in compliance with the IEEE 754-2008 decimal floating-point format [1]. Our contribution is to propose a correctly rounded, fast and reliable decimal multiple-precision interval decimal library. In the first part we will describe a short overview of the state of the art in decimal scientific libraries (Section 2). Then we will talk about the design of this library (Section 3), and next focus on the operations such as the conversion between decimal and binary (Section 4). Finally, after describing our testing methods and results (Section 5), we will finish by a conclusion and an outlook of the work that can still be done (Section 6).

2 State of the Art

There are numerous arithmetic libraries available that allow us to perform computation more accurately than the standard floating point arithmetic. We focus on two methods which provide a more reliable way to perform arithmetic operations: multiple-precision arithmetic and interval arithmetic. Each one has its advantages and drawbacks, and they can very well be combined together as multiple-precision interval arithmetic.

In this section we will shortly describe some of the interesting arithmetic libraries for our problem, and we will develop our motivations for the choices in the implementation.

2.1 Short Taxonomy of Arithmetic Libraries

The GNU MP, MPFR and MPFI libraries

When we talk about multiple-precision libraries, a lot of high level `Bignum` libraries in all kinds of languages like Python or Java come to mind. However, most of them rely on the C GNU Multiple Precision Arithmetic Library, or GMP [1] [6], which provides a fast and reliable arbitrary precision arithmetic on signed integers and rational numbers. GMP has a rich set of functions, and the functions have a regular interface. This means that each GMP function is defined the same way, as `mpz_function_name`(*output*, *inputs*). The memory available for the computation is the only practical limitation to the precision used by GMP operations.

The MPFR library [3] is the multiple-precision floating-point extension of GMP and mostly compliant with the IEEE 754 standard. It offers a larger set of functions, and ensures correctly rounded binary floating-point arithmetic. Each MPFR number is represented by a multiple-precision binary significand, an integer exponent, the sign and the expected output binary precision.

The MPFI library [11] is based on MPFR and provides multiple-precision floating-point arithmetic on intervals. In MPFI, the intervals are represented by two MPFR numbers as the lower and upper bound. The computation of those bounds is made possible by using directed rounding as parameter of the MPFR function. The correctness of those bounds is ensured by the correct rounding property of MPFR.

MPFR and MPFI follow the same regular interface and well defined semantics as GMP. They are low level libraries, written in C, designed to be fast and reliable.

INTLAB: MATLAB interval library

INTLAB [12] is an interval toolbox that is said to offer multiple-precision interval arithmetic for MATLAB. INTLAB supports real and complex intervals, but also vectors, full and sparse matrices over those.

INTLAB is designed to be very fast, especially on optimized algorithms for matrices with intervals. It also uses the MATLAB interactive programming interface.

The main propose of INTLAB is to produce reliable results fast while keeping the interface with the user at a high level. The user does not have to pay attention to rounding modes or the type of the variables. The interval arithmetic is enabled by the conversion of floating-point numbers with the conversion routine `str2intval`.

Jinterval for decimal intervals

When we think of fast computing, we mostly think about low level languages. However Java has its advantages, and as shown in [10] with the Jinterval library there is a way to compute interval arithmetic in an efficient way.

Jinterval performs interval arithmetic on rationals, with the `ExtendedRational` class. This choice of representation is motivated by the desire to build an algebraic system closed under arithmetic operations. Jinterval is a fast and efficient library, which ensures compliance to the IEEE 1788-2015 *Standard for Interval Arithmetic* [2].

Jinterval also provides several interval representations, such as classic intervals (defined in Section 3.3), Kaucher intervals [4][5], or complex rectangular or circular

[1]`https://gmplib.org`

intervals. With this flexibility in the choice of interval arithmetic for computations, the user can switch from one arithmetic to another if they are compatible.

Java BigInteger and BigDecimal

The IEEE 754-2008 [1] revision introduced the definition of decimal floating-point format, with the purpose of providing a basis for a robust and correctly rounded decimal arithmetic. Although there are a lot of libraries implementing decimal arithmetic with words of 64 bits, there are very few in multiple precision arithmetic, and none in interval arithmetic.

Two of those Java classes that provide multiple-precision integer and decimal arithmetic are `BigInteger`[2] and `BigDecimal`[3]. The latter relies on the former, as the `BigDecimal` type is defined by a `BigInteger` significand, an integer exponent and a second integer `scale`, that gives the number of digits to the right of the decimal point. The precision of the computation and the rounding mode are provided by a `MathContext` constructor. This representation has interesting characteristics but the arithmetic functions implemented in those classes are very limited. They implement the basic operations of addition, multiplication and square root, yet lacking the exponential, logarithm and trigonometric functions. That is why we did not direct our choice of implementation toward those classes.

2.2 Choice of GNU MP and MPFR

Studies of implementation of decimal arithmetic libraries have shown that one of the simplest and efficient way to implement decimal functions is to use their binary counterparts [7]. This method makes it possible to rely on the efficiency of the binary arithmetic and focus more effort only on some critical points.

Several criteria are taken into account for the choice of implementation of our decimal multiple-precision interval library. They depend on the two goals we want to achieve with the development of this library.

First we want our library to be correctly rounded in decimal and reliable. But to do so we do not want to put too much effort in the implementation of new decimal functions. That is why we choose to rely on the GMP and MPFR libraries to provide binary correctly rounded arithmetic.

Second we want our library to be fast. As we choose to rely on binary functions, we want them to be as fast as possible since we will add some conversion operations. Those operations are not inexpensive, hence the wish for a fast multiple-precision integer and floating-point arithmetic to rely on.

3 Design of a Decimal Multiple-Precision Interval Library

In this section, we will first set our notations for the rest of the article. Then we will define our decimal format, followed by a short overview of the interval arithmetic chosen for our implementation. To finish, we will talk about the high level design of our library.

[2] `https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html`
[3] `https://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html`

3.1 Some definitions

In the rest of the article, we will use the following notations to describe the different sets we work with. Let us define:

- $\mathbb{R}$ the set of real numbers,
- $\mathbb{F}_p^2 = \left\{m \cdot 2^E | E \in \mathbb{Z}, m \in \mathbb{Z}, E_{min} \leq E \leq E_{max}, 2^{p-1} \leq |m| \leq 2^p - 1\right\} \cup \{0\}$ the set of binary representable numbers of precision p,
- $\mathbb{F}_k^{10} = \left\{n \cdot 10^F | F \in \mathbb{Z}, n \in \mathbb{Z}, F_{min} \leq F \leq F_{max}, |n| \leq 10^k - 1\right\}$ the set of decimal representable numbers of precision k.

We also choose specific notation for the numbers in those sets, such that:

$$\begin{aligned}
&x, y, z \in \mathbb{R} \text{ are used for real numbers,}\\
&a, b, c \in \mathbb{F}_p^2 \text{ are used for binary numbers,}\\
&\alpha, \beta, \gamma \in \mathbb{F}_k^{10} \text{ are used for decimal numbers,}\\
&\varepsilon, \eta, \theta \text{ will be used to describe errors.}
\end{aligned}$$

3.2 Decimal Number Representation

Up to our knowledge, there is no low level library for decimal multiple-precision arithmetic. So we need to begin our work by creating such a library. To do this we need to create a multiple precision representation of a decimal number.

$$\alpha \coloneqq n \cdot 10^F, \alpha \in \mathbb{F}_k^{10}. \tag{1}$$

The decimal number α is defined by:

- n the significand is a GMP signed integer,
- F the exponent is a 64 bit signed integer,
- an additional information about the class of x (NaN, $\pm\infty$, ± 0 or decimal number)
- k the decimal precision in digits, the library will ensure that $1 \leq |n| \leq 10^k - 1$ at all times.

The IEEE 754-2008 standard [1] defines a decimal floating-point format. A decimal floating-point number might have multiple representations, the set of those representations being called a cohort. The quantum of a finite floating-point representation is the value of a unit in the last position of its significand. It is used to distinguish two representations of the same decimal floating-point.

The decimal format that we offer is compatible with the IEEE 754-2008 decimal floating-point format, but as it does not introduce the notion of quantum, it is not compliant.

3.3 Interval Arithmetic

In this paper, by an interval, we mean a closed interval[8], which can be defined as follows:

$$[\underline{x}, \overline{x}] \coloneqq \{x \in \mathbb{R} \text{ such that } \underline{x} \leq x \leq \overline{x} \text{ for some } \underline{x}, \overline{x} \in \mathbb{R}, \underline{x} \leq \overline{x}\}. \tag{2}$$

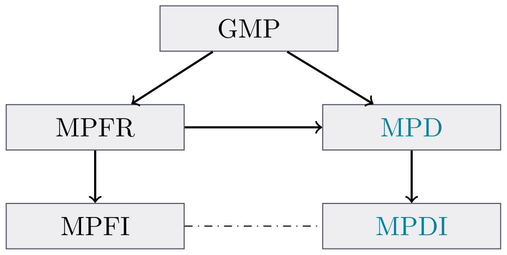

Figure 1: High Level Design, Architecture of the Libraries

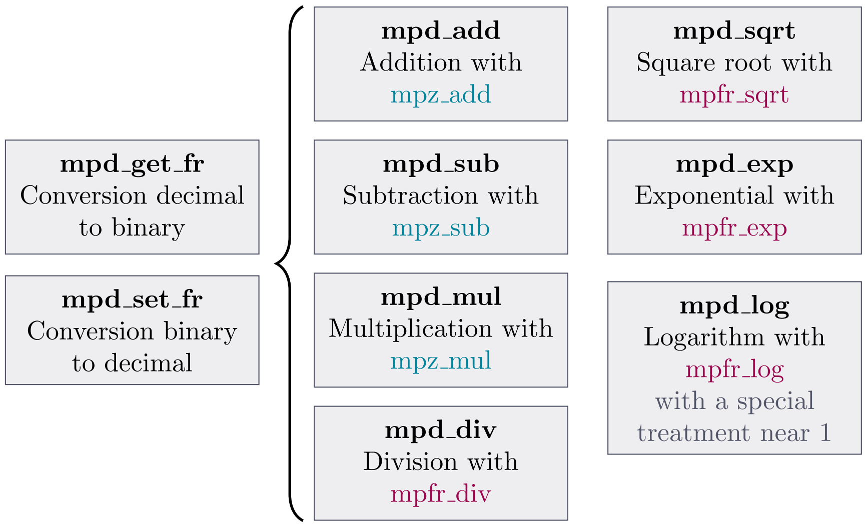
mpd_get_fr
Conversion decimal to binary
mpd_set_fr
Conversion binary to decimal
mpd_add
Addition with mpz_add
mpd_sub
Subtraction with mpz_sub
mpd_mul
Multiplication with mpz_mul
mpd_div
Division with mpfr_div
mpd_sqrt
Square root with mpfr_sqrt
mpd_exp
Exponential with mpfr_exp
mpd_log
Logarithm with mpfr_log with a special treatment near 1

Figure 2: Global architecture of MPD library

Interval arithmetic provides certified lower and upper bounds on the result of an operation. An operation on intervals is defined as:

$$[\underline{x}, \overline{x}] \circ [\underline{y}, \overline{y}] \supseteq \left\{ x \circ y \text{ such that } x \in [\underline{x}, \overline{x}], y \in [\underline{y}, \overline{y}] \right\}. \tag{3}$$

In our library, we represent an interval as two multiple precision decimal numbers:

$$[\underline{\alpha}, \overline{\alpha}] := \left\{ x \in \mathbb{R} \text{ such that } \underline{\alpha} \leq x \leq \overline{\alpha} \text{ for some } \underline{\alpha}, \overline{\alpha} \in \mathbb{F}_k^{10}, \underline{\alpha} \leq \overline{\alpha} \right\}. \tag{4}$$

With intervals, we can represent every number with finite bounds, even numbers that are not representable in decimal, nor binary such as $\frac{13}{17}$. The error introduced by the use of a finite floating-point representation is enclosed by the interval.

3.4 High Level Design

In order to implement our decimal multiple-precision interval library, we rely on the GMP library for the definition of our decimal format. Then, assuming a correctly rounded conversion function, we want to use binary functions to implement our library. That is why we use MPFR functions to compute some operations in binary.

Then MPDI library is built on MPD the same way MPFI relies on MPFR, as we can see from Figure 1. An interval is represented by the two decimal bounds, which are MPD numbers. The computation of those bounds is realized by the computation of the decimal MPD function with directed rounding modes.

Figure 2 illustrates the supported decimal operations, and their dependencies to GMP and MPFR. In each decimal operation, at least one binary operation is called. The correctness of the multiple-precision decimal arithmetic relies on a correctly rounded conversion algorithm.

4 Supported Operations

In this section, we go a little bit further into the description of our library, in particular of its basic operations, such as the multiplication and addition, and of the conversion algorithm. Then we will talk about elementary functions such as the logarithm or the exponential.

4.1 Basic Operations

It may seem strange to begin by describing the multiplication algorithm and not the addition or subtraction. But the reader will see that with our decimal format, the most simple operation is the multiplication.

4.1.1 Multiplication

The multiplication algorithm is clearly the simplest one, and it still gets simpler thanks to the format of our decimal representation. It consists of three steps:

$$\alpha \times \beta = n_\alpha \cdot 10^{F_\alpha} \times n_\beta \cdot 10^{F_\beta}, \alpha, \beta, \gamma \in \mathbb{F}_k^{10}. \quad (5)$$

1. Compute the result's exponent by adding the two exponents $F_\alpha + F_\beta$,
2. Compute a temporary significand by the multiplication of the two significands $n_\alpha \times n_\beta$, with the GMP multiplication function [6], such as

$$n_\alpha \times n_\beta = \texttt{gmp\_mul}(n_{temp}, n_\alpha, n_\beta)$$

3. Round the intermediary result n_{temp} at the required precision k and adjust the exponent accordingly.

Step three is made possible by our conversion algorithm that also serves the purpose of a rounding algorithm when the size of the number exceeds the precision required by the result.

4.1.2 Addition and Subtraction

The addition and subtraction algorithms are a bit trickier than the multiplication. Indeed, we cannot follow the basic binary floating-point algorithm that aligns the numbers according to their exponent by shifting the bits of their significand. In binary the shifting operation is inexpensive, as it comes back to memory manipulation.

To perform the same kind of alignment in decimal, we would need to use decimal division. However this operation is not cheap at all, that is why we must find a better way to do it.

As we describe it in section 4.1.1, the multiplication algorithm is the simplest we can have in decimal. In the worst case it only requires one rounding operation. This rounding is performed by our conversion algorithm described below.

Once the numbers are aligned by the multiplication by a power of ten, we can perform the addition or subtraction, without any cancellation. Then we round the result to the required precision k.

4.1.3 Conversion Algorithm

Let us denote, with $i \in \mathbb{N}^*$:

$\triangledown_i$ the rounding mode towards $-\infty$ with i bits of precision,

$\diamond_i$ any rounding mode with i bits of precision.

We now describe the major points of the conversion algorithm. It is presented as a conversion from binary to decimal but the same principles are applied for the conversion from decimal to binary.

This algorithm corresponds to the `mpd_set_fr` function, or the function that takes a MPFR binary floating point number and converts it into a MPD decimal number.

We want to convert the binary a into the decimal α such that:

$$a = m \cdot 10^E \text{ convert into } \alpha = n \cdot 10^F, a \in \mathbb{F}_p^2, \alpha \in \mathbb{F}_k^{10}. \tag{6}$$

Algorithm 1 Binary to decimal conversion from MPFR to MPD

```
Require:
    $a$: the MPFR such that $a = m \cdot 10^E$
    $k$: the decimal precision expected for the output
Ensure:
    $\alpha$: the MPD such that $\alpha = n \cdot 10^F$

 1: procedure MPD_SET_FR($a, \alpha$)
 2:     Compute the exponent $F$ as described in Equation (7)
 3:     Compute the intermediary precision $p'$ as described in Equation (8)
 4:     Compute $r = \diamond_{p'}(a \times \diamond_{p'}(10^{-F}))$ with MPFR functions
 5:     if $r$ is inexact then
 6:         Table's Maker Dilemma: $r \in [r_{low}, r_{high}]$ with $r_{low}, r_{high} \in \mathbb{F}_{p'}^2$
 7:     else
 8:         $r$ is exact: convert $r$ into $n$ a GMP integer
 9:     Convert $[r_{low}, r_{high}]$ into $[n_{low}, n_{high}]$, two GMP integers of precision $k$
10:     if $n_{low} = n_{high}$ then
11:         One of the bound is the signed multiple-precision integer $n$
12:     else
13:         Increase the precision $p'$ and goto line 4
```

The computation of the exponent can be done with a fixed binary precision because we defined it as a 64-bit integer. Hence we have

$$F = \lfloor t_{66} - (k-1) \rfloor \text{ with } t_{66} = \triangledown_{66}(\log_{10}(|x|)) \in \mathbb{F}_{66}^2. \tag{7}$$

All those operations are easily done with MPFR. We use 66 bits of precision because we only need two more bits to ensure that F, which is a 64-bit signed integer, is correct.

The first step to compute the significand is to compute an estimation p' of the binary precision needed in the general case to compute a decimal number of precision k, such that:

$$p' = k \cdot \log_2(10) + c, \tag{8}$$

where $c > 0$ is a constant allowing p' to be a slight overestimation of the binary equivalent of the output precision k, to perform internal calculation. In practice, we choose $c = 4$.

We then compute the approximated result with the binary precision p'. When performing an approximated computation, the rounding of the result can fall under the Table Maker's Dilemma [9]. It consists in finding for an approximation of a computed result the correct rounding in a given precision such that the last bit is the same as the exact result. In some cases, achieving the correct rounding can be hard.

In our algorithm, to overcome this dilemma, we use what is called a Ziv loop [13]. It consists in computing a result with more precision than the required precision for the output and deciding whether to round this result toward on or the other closest decimal number. To make that decision, the result is represented by an interval and compared with the midpoint between the two numbers. If the interval does not contain the midpoint, then the rounding is easy, else the computation of the interval must be done with a higher precision.

We can prove the termination and the correctness of this algorithm. For the termination, the computation of Equation Line 4 in the algorithm with a greater precision p' either provides new information on the computed digits, or is exact, and in that case the computation of the interval is not necessary. For the correctness, we can easily prove that there is a sufficiently large interval to manage the inherent error of the number we try to round. The trick is on one hand to take the interval large enough to ensure the correct rounding and on the other hand the smallest possible to minimize the computation time. We have shown that this corresponds to an interval of three binary representable number after the number we want to round in both directions.

The decimal to binary conversion is essentially the same, except the computation of the result is not separated in exponent and significand. We only compute

$$r = \diamond_{p'}(n \times \diamond_{p'}(10^F)). \tag{9}$$

Then we perform the same Ziv loop as for the binary to decimal conversion, to round the result according to the Table Maker's Dilemma.

4.2 Elementary Functions

4.2.1 Some Notation and Discussion of Error Propagation

To go further in building this arithmetic library, we need to take a moment to analyze the propagation of the error. Indeed, the computation of transcendental functions such as the exponential or the logarithm requires performing several conversions between decimal and binary, and thus introduces an error that will propagate through the algorithm.

There are three kinds of error that can occur during the general case of the computation of a decimal function $D\colon \mathbb{F}_k^{10} \to \mathbb{F}_k^{10}$ approximating a real function $f\colon \mathbb{R} \to \mathbb{R}$.

First the conversion error ε from a decimal $\alpha \in \mathbb{F}_k^{10}$ to a binary $a \in \mathbb{F}_p^2$:

$$\circ_2(\alpha) = a \cdot (1+\varepsilon) \text{ with } |\varepsilon| \le 2^{-p}. \tag{10}$$

Second the approximation error θ of the binary function B such that:

$$\begin{aligned} B\colon \mathbb{F}_p^2 &\to \mathbb{F}_p^2, \\ a &\mapsto f(a)(1+\theta). \end{aligned} \tag{11}$$

If B provides correct rounding to the nearest, then we have $|\theta| \le 2^{-p}$.

And third the conversion error η from binary back to decimal:

$$\circ_{10}(a) = \alpha \cdot (1+\eta) \text{ with } |\eta| \le 10^{-k}. \tag{12}$$

To sum-up, we can describe the decimal function D as follows:

$$\begin{aligned} D(\alpha) &= \circ_{10}(B(\circ_2(\alpha))) \\ &= B(a\cdot(1+\varepsilon))(1+\eta) \\ D(\alpha) &= f(a\cdot(1+\varepsilon))(1+\eta)(1+\theta) \end{aligned} \tag{13}$$

with $\alpha \in \mathbb{F}_k^{10}, a \in \mathbb{F}_p^2,$ and $\varepsilon, \theta, \eta \in \mathbb{R}$.

The errors η and θ are easy to handle. It is the first conversion error ε that gets amplified during the computation of D. Hence we have $D(\alpha) = f(a\cdot(1+\varepsilon))$.

We can approximate the function $D(\alpha)$ by $f(a\cdot(1+\varepsilon)) = f(a+a\varepsilon)$. Let n be an integer, if f is n times differentiable at the point $a \in \mathbb{F}_p^2 \subset \mathbb{R}$, then applying the Taylor-Lagrange formula, there exists a real number ξ between a and $a(1+\varepsilon)$ such that:

$$\begin{aligned} f(a+a\varepsilon) &= f(a) + f'(a)\frac{a\varepsilon}{1!} + \cdots + f^{(n)}(a)\frac{(a\varepsilon)^n}{n!} + \frac{f^{(n+1)}(\xi)}{(n+1)!}(a\varepsilon)^{n+1} \\ &= f(a)\left(1 + \frac{af'(a)}{f(a)}\varepsilon + \cdots + \frac{a^n f^{(n)}(a)}{n! f(a)}\varepsilon^n + \frac{a^{n+1} f^{(n+1)}(\xi)}{(n+1)! f(a)}\varepsilon^{n+1}\right). \end{aligned} \tag{14}$$

As computing the first term as an approximation of the function is enough to ensure the precision required by the output, we can easily overlook the higher order terms. In that case the stability of the error is linked to the condition number of the function, that is the value quantifying how much the output value of the function is affected by small changes in the input argument. We can then say that if the condition number is bounded, the propagation of the error is not important. However in case it is not, we will have to find some methods to compute the function in another way.

4.2.2 Exponential

Let us study the case of the exponential function. We want to bound the conversion error by following the protocol described in Section 4.2.1:

$$\begin{aligned} \exp(\alpha) &= \exp(a\cdot(1+\varepsilon)), \\ &= \exp(a)\cdot\exp(a\varepsilon), \\ &= \exp(a)\cdot(1+\underbrace{\exp(a\varepsilon)-1}_{\varepsilon'}). \end{aligned} \tag{15}$$

We have then the error such as:

$$\begin{aligned} \varepsilon' &= \exp(a\varepsilon) - 1, \\ 1 + \varepsilon' &= \exp(a\varepsilon), \\ \log(1+\varepsilon') &= a\varepsilon, \\ \varepsilon &= \frac{\log(1+\varepsilon')}{a}. \end{aligned} \tag{16}$$

This equations shows us that we can bound ε such that $|\varepsilon| \leq 2^{-p}$ for every value $a \in \mathbb{F}_p^2$. This implies that using only the conversion, computation and convert back method, we can implement the exponential function in decimal. If the error increases with the exponential, it is always negligible towards the precision wanted for the result.

4.2.3 Logarithm

The case of the logarithm is a bit trickier. To study it, we can write the logarithm as follows:

$$\begin{aligned} \log(\alpha) &= \log(a \cdot (1+\varepsilon)) \\ \log(\alpha) &= \log(a) \cdot \left(1 + \frac{1}{\log(a)} \cdot \frac{\log(1+\varepsilon)}{\varepsilon} \cdot \varepsilon\right) \end{aligned} \tag{17}$$

In this equation, we can see that the error ε is related to $\frac{1}{\log(a)}$. This means that whenever this quantity is unbounded, the error grows tremendously.

The point $\alpha = 1$ is manually defined by $\log(\alpha) = 0$. But near that point, the error produced by the conversion is amplified by the computation of the logarithm. This means that hence having a correctly rounded result with our conversion algorithm will take exponential time.

But for the logarithm function, we can avoid this problem quite easily by calling the MPFR function `mpfr_log1p` near 1. To do so, we need first to compute $\beta = \alpha - 1$ in decimal. Near $\alpha = 1$ this subtraction is exact, this can be proven thanks to the Sterbenz's theorem[9]. Then we use `mpfr_log1p` on the binary counterpart b of β. The function will compute $\circ_2(\log(1+b)) = \circ_2(\log(1+a-1)) = \circ_2(\log(a))$. In the end we will have the result we are looking for.

This solution is not always possible, and in most cases a function does not have only one point of amplification of the conversion error, e.g., the trigonometric function such as sin and cos, which require some more work to compute in our decimal multiple-precision interval library.

5 Experimental Results

In this section we will discuss our experimental approach and results by first talking a bit more about our implementation. In a second part the problem of correctness of the result will be addressed, and finally the timings of our libraries.

5.1 Correctness Testing

To perform the validation of the correctness of our interval library, we first need to certify the correct rounding of our decimal multiple-precision library. That ultimately

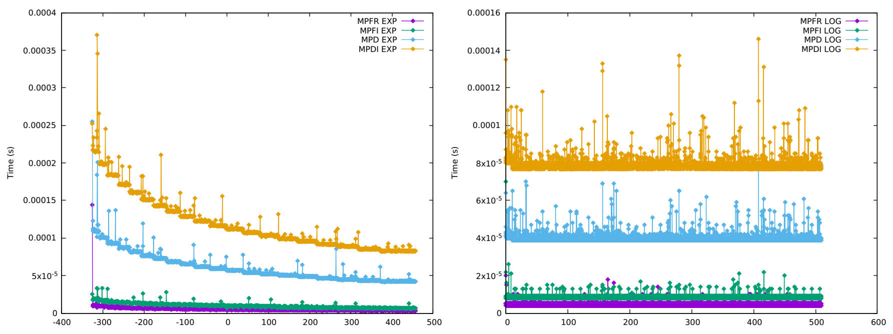

Figure 3: Timings of exp and log for a precision $k = 26$ digits

boils down to set a decimal number as entry of the test, and verify the correctness of the computed result.

This approach raises two issues: what points in the functions to test, and how to verify the correctness of the multiple-precision decimal result. To do so we need to determine which input we want to test, with which precision, and also the expected output.

The MPFR test frame is very detailed and optimized by bug reports and user experiences. Hence we used it as inspiration for our MPD test frame, in particular the input values used to test specific sensible points of the functions to validate our algorithms.

In MPFR, the results given for the verification of the validation of the tests are decimals converted in binary with the `mpfr_set_str` conversion function. To test our MPD library, we cannot put the same output as MPFR for a given entry, as MPFR ensures a binary number of bits correctly rounded, and MPD ensures it for a decimal number of digits. This implies that with the same input, the decimal output result is more accurate in MPD as in MPFR.

The tests of MPDI are essentially the same as the MPFI test framework, as it mimics its behavior. The same consideration than with MPD and MPFR tests are taken into account for the process of validation of the correctness of the result.

5.2 Performance Testing

The tests are performed on a Intel® Core$^{\text{TM}}$i5-3320M CPU @ 2.60GHz $\times$ 4, with the operating system Ubuntu 16.04.1 LTS 64 bits. The version of the libraries used for our implementation are GMP 6.1.1 and MPFR 3.1.5. The comparison of the timing results is made with the libraries MPFR 3.1.5 and MPFI 1.5.1. All test programs are compiled with gcc version 5.4.0.

5.2.1 Implementation Details

Apart from the arithmetical functions described in Section 4, other basic functions have been implemented in our MPD and MPDI libraries. Among them there is the comparison, implemented for both the decimal library and the interval library. These functions also use the conversion algorithm.

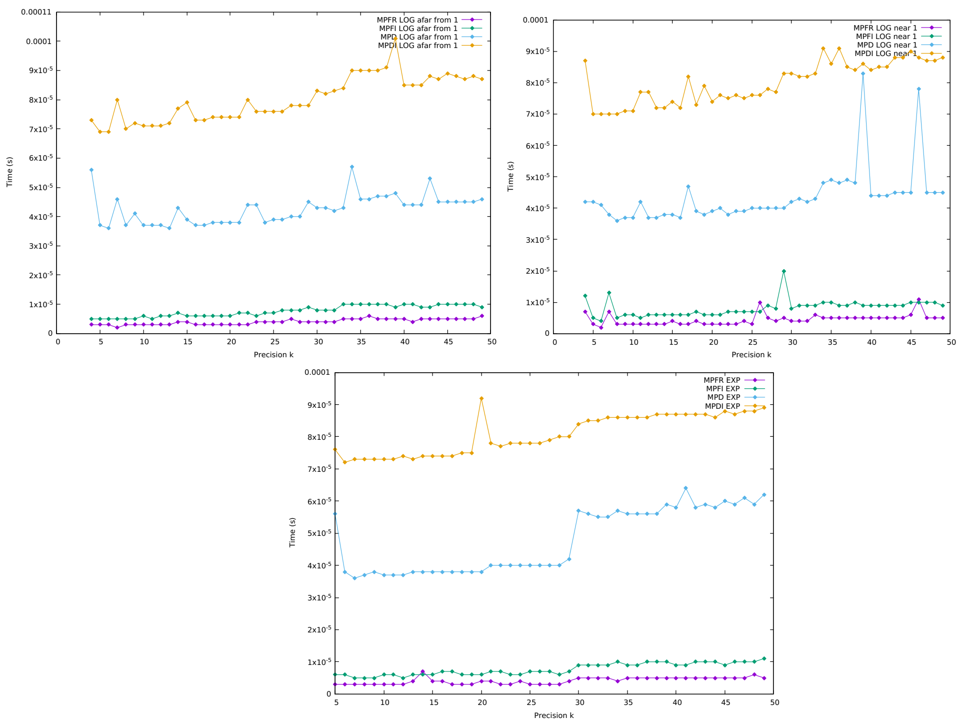

Figure 4: Timings of exp and log in function of the precision k for a given value

So basically, in almost all of our functions, the conversion algorithm is called at least once. This algorithm calls both MPFR and GMP functions, and when the internal precision is not accurate enough to compute the expected precision, the computation is done again with more digits. This implies that we expect the MPD library to be more than twice slower than the MPFR library.

The MPDI library behaves as MPFI by computing two times the same algorithm with the bounds of the interval. It is then expected to be as much as twice slower than the MPD library.

5.2.2 Computation Time

To illustrate the computation time considerations of our MPD and MPDI libraries, we choose to compare the result of the exponential and the logarithm algorithms with their binary counterparts in MPFR and MPFI.

As expected, and for the reasons highlighted in Section 5.2.1, we can see in Figure 3 that the decimal algorithms are slower than the binary ones. This Figure illustrates the computation time in seconds for the exponential and the logarithm for a set of entries, with a required decimal output precision $k = 26$ digits, and the equivalent precision p for the binary algorithms. On this Figure, we can see that the MPD functions are in general four times slower than the binary ones, and as expected the MPDI algorithms are twice slower than MPD. We can see peaks in the graphs, that can be explained by the computation of an output number which needed more intermediary precision to achieve the expected precision $k = 26$.

We tried a different approach to measure the computation time of the algorithms, where we compare the computation time between decimal and binary algorithm in function of the output precision required, as illustrated in Figure 4. The overall conclusion is the same as before, the decimal algorithm is in general four times slower, and the interval one eight times. We separated the timing of the computation of the logarithm with an input close to 1 and afar from 1 but we do not see any difference in the computation time.

However, the increase in computation time is balanced by the fact that the decimal accuracy of the result is in general higher than with the binary counterparts.

6 Conclusion and Perspectives

In this paper we have presented MPDI, a decimal multiple-precision interval library written in C. This library is a correctly-rounded arithmetic for decimal arbitrary precision. It contains fast and reliable basic operations and some elementary functions. Furthermore the decimal representation is IEEE 754-2008 compatible, and the interval arithmetic is compliant with the new IEEE 1788-2015 *Standard for Interval Arithmetic*.

MPDI is build upon a decimal-multiple precision library MPD that rely on GMP and MPFR. This library is still under development, and new operations such as the trigonometric functions sin, cos and tan will be implemented soon. Another goal is to increase the performance of the different algorithms, by narrowing the use of the conversion algorithm and perform other optimizations.

Acknowledgement

We are very grateful to the anonymous referees for the detailed and very helpful review of our paper.

References

[1] IEEE P-754 Working Group for Floating Point Arithmetic. IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, August 2008.

[2] IEEE P-1788 Working Group for Interval Arithmetic. IEEE Standard for Interval Arithmetic. *IEEE Std 1788-2015*, pages 1–97, June 2015.

[3] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, 33(2), June 2007.

[4] Alexandre Goldsztejn. Modal intervals revisited, part 1: A generalized interval natural extension. *Reliable Computing*, 16:130–183, 2012.

[5] Alexandre Goldsztejn. Modal intervals revisited, part 2: A generalized interval mean value extension. *Reliable Computing*, 16:184–209, 2012.

[6] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.1.1 edition, 2016. `http://gmplib.org/`.

[7] John Harrison. Decimal transcendentals via binary. In *Proceedings of the 2009 19th IEEE Symposium on Computer Arithmetic*, ARITH '09, pages 187–194, Washington, DC, USA, 2009. IEEE Computer Society.

[8] Ramon E. Moore, R. Baker Kearfott, and Michael J. Cloud. *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009.

[9] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Boston, 2010.

[10] Dmitry Yu. Nadezhin and Sergei I. Zhilin. Jinterval library: Principles, development, and perspectives. *Reliable Computing*, 19(3):229–247, 2014.

[11] Nathalie Revol and Fabrice Rouillier. Motivations for an arbitrary precision interval arithmetic and the MPFI library. *Reliable Computing*, 11(4):275–290, 2005.

[12] Siegfried M. Rump. Intlab — interval laboratory. In Tibor Csendes, editor, *Developments in Reliable Computing*, pages 77–104, Dordrecht, 1999. Springer Netherlands.

[13] Abraham Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Trans. Math. Softw.*, 17(3):410–423, September 1991.