CR-LIBM: The evaluation of the exponential

C. Daramy, David Defour, Florent de Dinechin, Jean-Michel Muller

Laboratoire de l'Informatique du Parallélisme

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

CR-LIBM: The evaluation of the exponential

Catherine Daramy, INRIA
David Defour, LIP
Florent de Dinechin, LIP
Jean-Michel Muller, CNRS

Juillet 2003

Research Report N° 2003-37

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

CR-LIBM: The evaluation of the exponential

Catherine Daramy, INRIA
David Defour, LIP
Florent de Dinechin, LIP
Jean-Michel Muller, CNRS

Juillet 2003

Abstract

We present a new elementary function library, called CR-LIBM. This library implements the various functions defined by the Ansi99 C standard. It provides correctly rounded functions. When writing this library, our primarily goal was to certify correct rounding, and make it reasonably fast, and with a low utilisation of memory. Hence, our library can be used without any problem on real-scale problems.
We are also giving the proof and the elements to understand the implementation of the exponential function of CR-LIBM.

Keywords: Elementary Functions, Exponential, CRlibm, correct rounding.

Résumé

Nous présentons une nouvelle bibliothèque d'évaluation de fonctions élémentaires, appelée CR-LIBM. Cette bibliothèque implémente les différentes fonctions définies par le standard Ansi C99. Sa principale caracteristique est de fournir l'arrondi correct pour la double précision et les quatre modes d'arrondi. Lors de l'écriture de cette bibliothèque, nos principaux objectifs étaient de certifier l'arrondi correct en ne dégradant pas les performances, et en limitant l'utilisation de mémoire. De ce fait, les fonctions de notre bibliothèque peuvent être utilisées sans problèmes dans des applications réelles.
Nous donnons également pour l'implémentation de l'exponentielle, la preuve pour certifier l'arrondi correct, et les éléments nécessaires pour comprendre les choix faits.

Mots-clés: Fonctions élémentaires, Exponentielle, CRlibm, arrondi correct.

CR-LIBM: The evaluation of the exponential*

Contents

*Authors may be reached via e-mail at {Catherine.Daramy, David.Defour, Florent.de.Dinechin, Jean-Michel.Muller}@ens-lyon.fr, This text is also available as a research report of the Institut National de Recherche en Informatique et en Automatique `http://www.inria.fr`.

1 Introduction

The need for accurate elementary functions is important in many critical programs. Methods for computing these functions include table-based methods[13, 26], polynomial approximations and mixed methods[7]. See the books by Muller[24] or Markstein[22] for recent surveys on the subject.

The IEEE-754 standard for floating-point arithmetic[16] defines the usual floating-point formats (single and double precision). It also specifies the behavior of the four basic operators ($+, -, \times, \div$) and the square root in four rounding modes (to the nearest, towards $+\infty$, towards $-\infty$ and towards 0). Its adoption and widespread use have increased the numerical quality of, and confidence in floating-point code. In particular, it has improved *portability* of such code and allowed construction of *proofs* on its numerical behavior. Directed rounding modes (towards $+\infty$, $-\infty$ and 0) also enabled efficient *interval arithmetic*[23, 17].

However, the IEEE-754 standard specifies nothing about elementary functions, which limits these advances to code excluding such functions. Currently, several options exist: on one hand, we can use today's mathematical libraries that are efficient but without any warranty on the correctness of the results. When strict guarantees are needed, some multiple-precision packages like MPFR [3] offer correct rounding in all rounding modes, but are several orders of magnitude slower than the usual mathematical libraries for the same precision. The recently released IBM Ultimate Math Library[1] claims to offer correct rounding to the nearest, and this library is both portable and fast, if bulky. However, for reasons detailed below, this claim is not proven. Besides, this library still lacks directed rounding modes needed for interval arithmetic, and has other drawbacks that we analyze in the sequel.

The purpose of this paper is to show that the combination of several recent advances allows us to design a correctly rounded mathematical library which is fast enough to replace the existing libraries, at a minor cost in terms of performance and resources. Section 2 presents the context of this library. Section 3 presents the state of the library. Section 4 give some notations and results use in the sequel of this paper. Section 5 to 10 describe the exponential and prove the correct rounding of the exponential function. Section 11 analyzes this function, and shows that it is comparable in size and speed to other mathematical libraries.

2 A methodology for efficient correctly-rounded functions

2.1 The Table Maker's Dilemma

With a few exceptions, the image y of a floating-point number x by a transcendental function f is a transcendental number, and can therefore not be represented exactly in standard numeration systems. The only hope is to compute the floating-point number that is closest to (resp. immediately above or immediately below) the mathematical value, which we call the result *correctly rounded* to the nearest (resp. towards $+\infty$ or towards $-\infty$).

It is only possible to compute an approximation $\hat{y}$ to the real number y with precision ε. This ensures that the real value y belongs to the interval $[\hat{y} - \varepsilon, \hat{y} + \varepsilon]$. Sometimes however, this information is not enough to decide correct rounding. For example, if $[\hat{y} - \varepsilon, \hat{y} + \varepsilon]$ contains the middle of two consecutive floating-point numbers, it is impossible to decide which of these two numbers is the correctly rounded to the nearest of y. This is known as the Table Maker's Dilemma (TMD).

2.2 The onion peeling strategy

A method described by Ziv [27] is to increase the precision ε of the approximation until the correctly rounded value can be decided. Given a function f and an argument x, the value of $f(x)$ is first evaluated using a quick approximation of precision ε_1. Knowing ε_1, it is possible to decide if rounding is possible, or if more precision is required, in which case the computation is restarted using a slower approximation of precision ε_2 greater than ε_1, and so on. This approach makes sense even in terms of average performance, as the slower steps are rarely taken.

However there was until recently no practical bound on the termination time of such an algorithm. This iteration has been proven to terminate, but the actual maximal precision required in the worst case is unknown. This might prevent using this method in critical application.

2.3 An overview of available mathematical libraries

Many high-quality mathematical libraries are freely available, including *fdlibm*, written by Sun[2] and *libultim* written by IBM[1], which are portable assuming IEEE-754 arithmetic, and processor-specific libraries by Intel[15, 4] and HP[22, 21] among other. Operating systems often include several mathematical libraries, some of which are derivatives of one of the previous.

Among these libraries, two offer correct correct rounding:

- The*libultim* library also called MathLib, is developed at IBM by Ziv and others [1]. It provides correct rounding, under the assumption that 800 bits are enough in all case. This approach suffers two weaknesses. The first is the absence of proof that 800 bits are enough: all there is is a very high probability. The second is that, as we will see in the sequel, for challenging cases, 800 bits are much of an overkill, which can increase the execution time up to 20,000 times a normal execution. This will prevent such a library from being used in real-time applications. Besides, to prevent this worst case from degrading average performance, there is usually some intermediate levels of precision in MathLib's elementary functions, which makes the code larger, more complex, and more difficult to prove.

  In addition this library provides correct rounding only to nearest. This is the most used rounding mode, but it might not be the most important as far as correct rounding is concerned: correct rounding provides a precision improvement over current mathematical libraries of only a fraction of a unit in the last place *(ulp)*. Conversely, the three other rounding modes are needed to guarantee intervals in interval arithmetic. Without correct rounding in these directed rounding modes, interval arithmetic looses up to one *ulp* of precision in each computation.

- *MPFR* is a multiprecision package safer than *libultilm* as it uses arbitrary multiprecision. It provides most of elementary functions for the four rounding modes defined by the IEEE-754 standard. However this library is not optimized for double precision arithmetic. In addition, as its exponent range is much wider than that of IEEE-754, the subtleties of denormal numbers are difficult to handle properly using such a multiprecision package.

3 The Correctly Rounded Mathematical Library

We have designed our own library called *crlibm* (correctly rounded mathematical library). It is based on the work of Lefèvre[20, 19] who computed the worst-case ε required for correctly rounding several functions in double-precision over selected intervals in the four IEEE-754 rounding modes. For example, he proved that 157 bits are enough to ensure correct rounding of the exponential function on all of its domain for the four IEEE-754 rounding modes.

3.1 Two steps are enough

Thanks to such results, we are able to guarantee correct rounding in two iterations only, which we may then optimize separately. The first of these iterations is relatively fast and provides between 60 and 80 bits of accuracy (depending on the function), which is sufficient in most cases. It will be referred throughout the paper as the **Quick** phase of the algorithm. The second phase, referred to as the **Accurate** phase, is dedicated to challenging cases. It is slower but has a reasonably bounded execution time, tightly targeted at Lefèvre's worst cases.

Having a proven worst-case execution time lifts the last obstacle to a generalization of correctly rounded transcendentals. Besides, having only two steps allows us to publish, along with each function, a proof of its correctly rounding behavior.

3.2 Portable IEEE-754 FP for fast first step

The computation of a tight bound on the approximation error of the first step (ε_1) is crucial for the efficiency of the onion peeling strategy: overestimating ε_1 means going more often than needed through the second step. As we want the proof to be portable as well as the code, our first steps are written in strict IEEE-754 arithmetic. On some systems, this means preventing the compiler / processor combination to use advanced floating-point features such as fused multiply-and-add or extended double precision. It also means that the performance of our portable library will be lower than optimized libraries using these features.

To ease these proofs, our first steps make wide use of classical, well proven techniques. In particular, when a result is needed in a precision higher than double precision (as is the case of $\hat{y_1}$, the result of the first step), it is represented as as the sum of two floating-point numbers. There are well-known algorithms for computing on such sums (for instance Sterbenz' lemma, the Fast2Sum algorithm, the Dekker algorithm[18]) with mechanically checked proofs.

A sequence of simple tests on $\hat{y_1}$ allows to decide whether to go for the second step. The sequence corresponding to each rounding mode is shared by most functions and has also been carefully proven.

3.3 Software Carry-Save for an accurate second step

For the second step, we designed an ad-hoc multiple-precision library called Software Carry-Save library *(scslib)* which is lighter and faster than other available libraries for this specific application [11, 10]. This choice is motivated by considerations of code size and performance, but also by the need to be independent of other libraries: Again, we need a library on which we may rely at the proof level. This library is independent from the mathematical library and distributed separately [5].

3.4 Current state of *crlibm*

The library `crlibm` *(correctly rounded mathematical library)* currently offers accurate parts for the exponential, logarithm in radix 2, 10 and e, sine, cosine, tangent, arctangent, plus trigonometric argument reduction. The first quick part and its proof have only been written for the exponential thus far. The difficulty is to prove both the algorithm and the C program. The proof relies heavily on several shared lemmas, assuming the good behavior of the system composed of the compiler and the processor. Another difficulty is that performance is important. Therefore many parts of this proof could be done only by hand.

4 Notations and useful results

Throughout the paper, we will note $+$, $-$ and $\times$ the usual mathematical operations, and $\oplus$, $\ominus$ and $\otimes$ the corresponding floating-point operations in IEEE-754 double precision, in the IEEE-754 *round to nearest* mode. To characterize the error we will use the following definition :

Definition 1 (ε_n) *For any integer n, we will define by ε_n a value α such that:*

$$|\alpha| \leq 2^n$$

For a floating-point number x, we will classically denote $\text{ulp}(x)$ the value of the least significant bit of its mantissa.

We will make use of the following well-known results:

Theorem 1 (Sterbenz Lemma [25, 14]) *If x and y are floating-point numbers, and if $y/2 \le x \le 2y$ then $x \ominus y$ is computed exactly, without any rounding error.*

A double precision floating-point number is coded on 64 bits, that is two times the size of an integer, usually represented with 32 bits in current processors. The order in which the two 32 bits words are stored in memory depends on the architecture. An architecture is said *Little Endian* if the lower part of the number is stored in memory at the smallest address; x86 processor use this representation. Conversely, an architecture with the higher part of the number stored in memory at the smallest address is said *Big Endian*; PowerPC processor use this representation.

The following code extracts the upper and lower parts of a double precision number x in a classical and relatively portable way.

Listing 1: Extract upper and lower part of a double precision number x

```
/* LITTLE_ENDIAN/BIG_ENDIAN are define by the user or   */
/* automatically by tools such as autoconf/automake.    */

#ifdef LITTLE_ENDIAN
#define HI(x) *(1+(int*)&x)
#define LO(x) *(int*)&x
#elif BIG_ENDIAN
#define HI(x) *(int*)&x
#define LO(x) *(1+(int*)&x)
#endif
```

The previous code is also efficient on many architectures where FP and integer pipelines are hermetic, requiring conversions from one format to the other to be done through memory. There are probably architectures where a more efficient implementation can be found.Our code uses exclusively the previous two function for converting FP to integer and back, wich ensures a quick implementation of such an architecture-specific optimization.

Theorem 2 (Fast2sum algorithm [18]) *For a and b two floating-point numbers, the following method computes two floating-point numbers s and r, such that $s + r = a + b$ exactly, and s is the floating-point number which is closest to $a + b$.*

Listing 2: Fast2SumCond

```
#define Fast2SumCond(s, r, a, b) \
        {double z, _a=a, _b=b;    \
         s = _a + _b;             \
         if ((HI(_a)&0x7FF00000) > (HI(_b)&0x7FF00000)){   \
           z = s - _a;            \
           r = _b - z;            \
         }else {                  \
           z = s - _b;            \
           r = _a - z;            \
         }
```

This algorithm requires 3 floating-point additions, 2 masks and 1 test over integer.

In the case we know that the exponent of a is greater than the one of b, then the previous algorithm to perform an exact addition of 2 floating-point numbers becomes:

Listing 3: Fast2Sum

```
#define Fast2Sum(s, r, a, b)      \
        {double z, _a=a, _b=b;    \
         s = _a + _b;             \
         z = s - _a;              \
         r = _b - z; }
```

The cost of this algorithm is 3 floating-point additions.

Theorem 3 (double multiplication[12, 18]) *Let a and b be two floating-point numbers, with $p \geq 2$ the size of their mantissa. Let $c = 2^{\frac{\lceil p \rceil}{2}} + 1$. The following method computes the two floating-point numbers $r1$ and $r2$ such that $r1 + r2 = a + b$ with $r1 = a \otimes b$ in the case $p = 53$ (double precision):*

Listing 4: DekkerCond

```
void inline DekkerCond(double *r1, double *r2, double a, double b){
  double two_m53 = 1.1102230246251565404e-16;  /* 0x3CA00000, 0x00000000 */
  double two_53  = 9007199254740992.;          /* 0x43400000, 0x00000000 */
  double c       = 134217729.;                 /* 0x41A00000, 0x02000000 */
  double u, up, u1, u2, v, vp, v1, v2, r1, r2;

  if (HI(a)>0x7C900000) u = a*two_m53;
  else                  u = a;
  if (HI(b)>0x7C900000) v = b*two_m53;
  else                  v = b;

  up = u*c;         vp = v*c;
  u1 = (u-up)+up;   v1 = (v-vp)+vp;
  u2 = u-u1;        v2 = v-v1;

  *r1 = u*v;
  *r2 = (((u1*v1-*r1)+(u1*v2))+(u2*v1))+(u2*v2)

  if (HI(a)>0x7C900000) {*r1 *= two_e53; *r2 *= two_53;}
  if (HI(b)>0x7C900000) {*r1 *= two_e53; *r2 *= two_53;}
}
```

We have to test a and b before and after the core of the algorithms in order to avoid overflow by multiplying by c. The global cost in the worst case is 4 tests over integers, 10 floating-point additions and 13 floating-point multiplications.

If we are know that a and b are less then 2^{970} we can skip this test, and get the following algorithm:

Listing 5: Dekker

```
void inline Dekker(double *r1, double *r2, double u, double v){
  double c       = 134217729.;                 /* 0x41A00000, 0x02000000 */
  double up, u1, u2, vp, v1, v2;

  up = u*c;         vp = v*c;
  u1 = (u-up)+up;   v1 = (v-vp)+vp;
  u2 = u-u1;        v2 = v-v1;

  *r1 = u*v;
  *r2 = (((u1*v1-*r1)+(u1*v2))+(u2*v1))+(u2*v2)
}
```

which reduces the cost of this algorithm to 10 floating-point additions and 7 floating-point multiplications.

It should be noted that the availability of fused multiply-and-add (FMA), with only one rounding, on architectures like PowerPC and IA-64, allows the implementation of the Dekker algorithm in only two operations: *r1 = u*v ; *r2 = FMA(u*v-r1) ; Again, this is an architecture-dependent optimization.

Theorem 4 (Conversion from floating-point to integer [6]) *The following algorithm convert a floating-point number d into an integer i with rounding to nearest mode.*

Listing 6: Solution 2

```
#define DOUBLE2INT(i, d) \
  {double t=(d+6755399441055744.0); i=LO(t);}
```

This algorithm add the constant $2^{52} + 2^{51}$ to the floating-point number to put the integer part of x, in the lower part of the floating-point number. We use $2^{52} + 2^{51}$ and not 2^{52}, because the value 2^{51} is used to contain possible carry propagations with negative numbers.

5 Overview of the method for the exponential

We are now going to present and proof the correct rounding of the evaluation scheme chosen for the exponential within *crlibm*. We will use a property deduced from the enumeration of worst cases done for the exponential function by Lefèvre [19]

Property 1 (Correct rounding of the exponential) *Let y be the result of the exponential of a floating-point number x in double precison. Let y^* be an approximation of y such that the distance between mantissa of y and y^* is bounded by ε.*

If $\varepsilon \leq 2^157$ then rounding y^* is equivalent to rounding y for the four rounding modes.

We have done the evaluation of the exponential in two steps. First, we use the *quick* phase of the algorithm to get an approximation good to 68 bits of the result. Then we perform a test to check whether we need to use the *accurate* phase, based on multiprecision operators from SCSlib [5] .

To increase the trust in the code, with have included constants in hexadecimal format (big endian only for concision). However to help the reader we are giving the corresponding decimal values.

6 Quick phase

Here is the general scheme chosen for the first step of the evaluation of the exponential:

1. **"Mathematical" range reduction**
   We compute the reduced argument
   $$(r\_hi + r\_lo) \in [-\ln(2)/2, +\ln(2)/2]$$
   such that:
   $$x = k.\ln(2) + (r\_hi + r\_lo)$$
   therefore
   $$\exp(x) = \exp(r\_hi + r\_lo).2^k$$

2. **Tabular range reduction**
   Let $index\_flt$ be the first 8 bits of $(r\_hi + r\_lo)$ and $(rp\_hi + rp\_lo) = (r\_hi + r\_lo) - index\_flt$, such that $(rp\_hi + rp\_lo) \in [-2^{-9}, +2^9]$. We have
   $$\exp(r\_hi + r\_lo) = \exp(index\_flt) \times \exp(rp\_hi + rp\_lo)$$
   where $\exp(index\_flt) = (ex\_hi + ex\_lo)$ will be looked up in a table.

3. **Polynomial evaluation**
   We evaluate the polynom P_r of degree 3 such that:
   $$exp(rp\_hi + rp\_lo) \approx 1 + (rp\_hi + rp\_lo) + \frac{1}{2}.(rp\_hi + rp\_lo)^2 + (rp\_hi + rp\_lo)^3.(P\_r)$$
   with $P\_r = c_0 + c_1.rp\_hi + c_2.rp\_hi^2 + c_3.rp\_hi^3$ and $rp\_hi \in [-2^{-9}, +2^9]$

4. **Reconstruction**
   $$\begin{aligned}\exp(x) \quad = \quad & 2^k.(ex\_hi + ex\_lo). \\ & (1 + (rp\_hi + rp\_lo) + \tfrac{1}{2}.(rp\_hi + rp\_lo)^2 + (rp\_hi + rp\_lo)^3.P\_r).(1 + \varepsilon_{-68})\end{aligned}$$

6.1 Handling special cases

6.1.1 Methods to raise IEEE-754 flags

The IEEE standard requires, in certain cases, to raise flags and exceptions for the operators $+$, $\times$, $\div$, $\sqrt{}$. Therefore, it is legitimate to require the same for elementary functions. For portability, these exceptions and flags will be generated using the following techniques:

- **underflow**: The multiplication $\pm smallest \times smallest$ where $smallest$ correspond to the smallest denormal number,
- **overflow**: The multiplication $\pm largest \times largest$ where $largest$ correspond to the largest normal number,
- **division by zero**: The division $\pm 1.0/0.0$,
- **inexact**: The addition $(x + smallest) - smallest$ where x is the result,
- **invalid**: The division $\pm 0.0/0.0$.

6.1.2 Avoiding overflows and underflows

In the sequel of this paper, we will consider input numbers in the range $[u\_bound, o\_bound]$, where u_bound and o_bound are:

$$u\_bound = \triangle\left(\ln\left(\left(1-2^{-53}\right).2^{-1075}\right)\right) = -745.1332\ldots$$
$$o\_bound = \triangledown\left(\ln\left(\left(1-2^{-53}\right).2^{1024}\right)\right) = 709.7827\ldots$$

where $\triangle(x)$ and $\triangledown(x)$ respectively correspond to the rounding toward $+\infty$ and $-\infty$ of x.

In the rounding mode to nearest, the exponential of a number greater than o_bound is an overflow, whereas the exponential of a number less than u_bound is rounded to 0, and raises an inexact flag.

However, subtler under/overflow situations may arise in two cases, which we should avoid:

- An intermediate computation may raise an overflow although the final result is representable as an IEEE-754 floating-point number.
- In IEEE-754 arithmetic, when a result is between 2^{-1023} and 2^{-1074}, a gradual underflow exception arises to signal that the precision of the result is reduced in a drastic way.

In both cases, as we will show in the following, it is possible to avoid the exception by predicting that it will occur, and appropriately scaling the input number in the range reduction phase.

6.1.3 Rounding to nearest

Listing 7: Handling special cases in rounding to nearest

```
 static const union{int i[2]; double d;}
#ifdef BIG_ENDIAN
  _largest      = {0x7fefffff, 0xffffffff},
  _smallest     = {0x00000000, 0x00000001},
  _u_bound      = {0xC0874910, 0xD52D3052}, /* -7.45133219101941222107e+02 */
  _o_bound      = {0x40862E42, 0xFEFA39F0}; /*  7.09782712893384086783e+02 */
#else
  ...
#endif
#define largest    _largest.d
#define smallest   _smallest.d
#define u_bound    _u_bound.d
#define o_bound    _o_bound.d

unsigned int hx;

hx  = HI(x);
hx &= 0x7fffffff;

```

```
20 /* Filter special cases */
21 if (hx >= 0x40862E42){
22   if (hx >= 0x7ff00000){
23     if (((hx&0x000fffff)|LO(x))!=0)
24       return x+x;                            /* NaN */
25     else return ((hx&0x80000000)==0)? x:0.0; /* exp(+/-inf) = inf,0 */
26   }
27   if (x > o_bound) return largest*largest ;  /* overflow  */
28   if (x < u_bound) return smallest*smallest; /* underflow */
29 }
30
31
32 if (hx <= 0x3C900000) return 1.;             /* if (hx <= 2^(-54)) */
```

◇ *Proof.*

line 17	Put the high part of x in hx. (cf. prog. 1)
line 18	Remove the sign information within hx. It will make tests on special cases simpler.
line 21	Test equivalent to $if(\|x\| >= 709.7822265625)$. This test is true if $x > u\_bound$, $x < o\_bound$, $x = \pm inf$ or $x = NaN$. This test is performed with integers to make it faster.
line (22-24)	Test if $x = \pm inf$ or $x = NaN$ and give the corresponding results (exact $+\infty$ or 0).
line 27	Under the assumption that the compiler correctly translates the floating-point number we have $o\_bound = 390207173010335/549755813888$. If $x > o\_bound$ then $\exp(x) = +\inf$. The multiplication $largest * largest$ leaves to the compiler the generation of an overflow and the corresponding flags.
line 28	Under the assumption that the compiler correctly translates the floating-point number we have $u\_bound = -3277130554578985/4398046511104$. If $x < u\_bound$ then $\exp(x) = +0$. The multiplication $largest * largest$ leaves to the compiler the generation of an underflow and the corresponding flags.
line 32	Test equivalent to $if(\|x\| \leq 2^{-54})$. This test is performed with integers to make it faster and is valid because $x \notin \{NaN, \infty\}$. In addition, this test allows to handle cases when x is a denormal number. We have the following property:

<1> $|x| > 2^{-54}$ and $x \notin \{NaN, \infty\}$ •

Indeed, in rounding to nearest, if $|x| \leq 2^{-54}$ then $\exp(x) = 1.0$. This test prevents to encounter a denormal number in the rest of the program.

□

6.1.4 Rounding toward $+\infty$

Listing 8: Handling special cases in rounding toward $+\infty$

```
static const union{int i[2]; double d;}
#ifdef BIG_ENDIAN
 _largest      = {0x7fefffff, 0xffffffff},
 _smallest     = {0x00000000, 0x00000001},
 _u_bound      = {0xC0874910, 0xD52D3052}, /* -7.45133219101941222107e+02 */
 _o_bound      = {0x40862E42, 0xFEFA39F0}, /*  7.09782712893384086783e+02 */
 _two_m52_56 = {0x3CB10000, 0x00000000}; /*  2.35922392732845764840e-16 */
#else
 ...
#endif
#define largest     _largest.d
#define smallest    _smallest.d
#define u_bound     _u_bound.d
#define o_bound     _o_bound.d
#define two_m52_56 _two_m52_56.d


unsigned int hx;

```

```
hx   = HI(x);
hx &= 0x7fffffff;

/* Filter special cases */
if (hx >= 0x40862E42){
  if (hx >= 0x7ff00000){
    if (((hx&0x000fffff)|LO(x))!=0)
      return x+x;                             /* NaN */
    else return ((hx&0x80000000)==0)? x:0.0; /* exp(+/-inf) = inf,0 */
  }
  if (x > o_bound) return largest*largest;    /* overflow  */
  if (x < u_bound) return smallest*1.0;       /* 2^(-1074) */
}

if (hx < 0x3CA00000){                         /* if (hx <= 2^(-53)) */
  if (HI(x) < 0)
    return 1. + smallest;                     /* 1 and inexact */
  else
    return 1. + two_m52_56;                   /* 1 + 2^(-52) and inexact */
}
```

◇ *Proof.* This program is similar to the one used in rounding to nearest mode with the following exceptions:

- When $(x < u\_bound)$, in rounding toward $+\infty$, we have to return as result the smallest representable number (2^{-1074}) with the inexact flag raised.
- When $(|x| < 2^{-53})$, in rounding toward $+\infty$, we have to return as result 1.0 if $x < 0$ with the inexact flag raised or $1 + 2^{-52}$ with the inexact flag raised if $x > 0$.

□

6.1.5 Rounding toward $-\infty$

Listing 9: Handling special cases in rounding toward $-\infty$

```
static const union{int i[2]; double d;}
#ifdef BIG_ENDIAN
 _largest     = {0x7fefffff, 0xffffffff},
 _smallest    = {0x00000000, 0x00000001},
 _u_bound     = {0xC0874910, 0xD52D3052}, /* -7.45133219101941222107e+02 */
 _o_bound     = {0x40862E42, 0xFEFA39F0}, /*  7.09782712893384086783e+02 */
 _two_m52_56 = {0x3CB10000, 0x00000000}; /*  2.35922392732845764840e-16 */
#else
 ...
#endif
#define largest     _largest.d
#define smallest    _smallest.d
#define u_bound     _u_bound.d
#define o_bound     _o_bound.d
#define two_m52_56 _two_m52_56.d

unsigned int hx;

hx   = HI(x);
hx &= 0x7fffffff;

/* Filter special cases */
if (hx >= 0x40862E42){
  if (hx >= 0x7ff00000){
    if (((hx&0x000fffff)|LO(x))!=0)
      return x+x;                             /* NaN */
    else return ((hx&0x80000000)==0)? x:0.0; /* exp(+/-inf) = inf,0 */
  }
  if (x > o_bound) return largest*1.0;        /* (1-2^(-53))*2^1024  */
  if (x < u_bound) return smallest*smallest; /* underflow */
}

if (hx < 0x3CA00000){                         /* if (hx <= 2^(-53))    */
  if (HI(x) < 0)
    return 1. - two_m52_56;                   /* 1-2^(-52) and inexact */
  else
    return 1. + smallest;                     /* 1 and inexact */
}
```

◇ *Proof.* This program is similar to the one used in rounding to nearest mode with the following exceptions:

- When $(x > o\_bound)$, in rounding toward $-\infty$, we have to return as result the largest representable number $((1 - 2^{-53}).2^{1024})$ with the inexact flag raised.
- When $(|x| < 2^{-53})$, in rounding toward $-\infty$, we have to return as result $1.0 - 2^{-52}$ if $x < 0$ with the inexact flag raised or 1.0 with the inexact flag raised if $x > 0$.

□

6.1.6 Rounding toward 0

The exponential function is continuous and positive, therefore rounding toward 0 is equivalent to rounding toward $-\infty$.

6.2 The range reduction

6.2.1 First reduction step

The purpose of this first range reduction is to replace the input number $x \in [u\_bound, o\_bound]$ with two floating-point numbers r_hi, r_lo and an integer k such that:

$$x = k. \ln(2) + (r\_hi + r\_lo).(1 + \varepsilon)$$

with $|r\_hi + r\_lo| < \frac{1}{2}\ln(2)$

This "additive" range reduction may generate a cancellation if x is close to a multiple of $\ln(2)$. A method from Kahan based on continuous fractions (see Muller [24] pp 154) allows us to compute the worst cases for the range reduction. Examples of results are given in Table 2.

Interval	Worst cases	Number of bits lost
$]2^{1024}, 2^{1024}[$	$5261692873635770 \times 2^{499}$	$66, 8$
$[-1024, 1024]$	$7804143460206699 \times 2^{-51}$	$57, 5$

Table 2: Worst cases corresponding to the closest number multiple to $\ln(2)$, for the additive range reduction of the exponential. The maximum number of bits lost by cancellation is also indicated .

The interval $[u\_bound, o\_bound]$ on which we are evaluating the exponential is included within $[-1024, 1024]$. Therefore at most 58 bits can be cancelled during the subtraction of the closest multiple of $\ln(2)$ to the input number x.

Theorem 5 *The sequence of instructions of the program 10 computes two floating-point numbers in double precision r_hi, r_lo and an integer k such that*

$$r\_hi + r\_lo = (x - k \times \ln 2) + \varepsilon_{-69}$$

with k the closest integer to $x/\ln 2$.

Listing 10: First range reduction

```
static const union{int i[2]; double d;}
#ifdef BIG_ENDIAN
  _ln2_hi     = {0x3FE62E42, 0xFEFA3800}, /*  6.93147180559890330187e-01 */
  _ln2_me     = {0x3D2EF357, 0x93C76000}, /*  5.49792301870720995198e-14 */
  _ln2_lo     = {0x3A8CC01F, 0x97B57A08}, /*  1.16122272293625324218e-26 */
  _inv_ln2    = {0x3FF71547, 0x6533245F}; /*  1.44269504088896338700e+00 */
#else
  ...
#endif
#define ln2_hi      _ln2_hi.d
#define ln2_me      _ln2_me.d
```

```
50 #define ln2_lo      _ln2_lo.d
51 #define inv_ln2     _inv_ln2.d
52
53 double r_hi, r_lo, rp_hi, rp_lo;
54 double u, tmp;
55 int k;
56
57 DOUBLE2INT(k, x * inv_ln2)
58
59 if (k != 0){
60   /* r_hi+r_lo =  x - (ln2_hi + ln2_me + ln2_lo)*k */
61   rp_hi = x-ln2_hi*k;
62   rp_lo =  -ln2_me*k;
63   Fast2SumCond(r_hi, u, rp_hi, rp_lo);
64   r_lo = u - ln2_lo*k;
65 }else {
66   r_hi = x;  r_lo = 0.;
67 }
```

◇ *Proof.*

line (41-43)

<2> By construction: $ln2\_hi + ln2\_me + ln2\_lo = \ln(2)(1 + \varepsilon_{-140})$ •

<3> $|ln2\_hi| \leq 2^0 \quad |ln2\_me| \leq 2^{-44} \quad |ln2\_hi| \leq 2^{-86}$ •

<4> $ln2\_hi$ and $ln2\_me$ hold at most 42 bits of precision •

line 57 Put in k the closest integer of $x * inv\_ln2$. We use the property of DOUBLE2INT that converts a floating-point number in rounding to nearest mode (program 6, page 6). In addition k satisfies the following property:

<5> $\lfloor x \times inv\_ln2 \rfloor \leq k \leq \lceil x \times inv\_ln2 \rceil$ et $|k| \leq \frac{x}{2} \times inv\_ln2$ •

We have seen in Section 6.1.2: $-745.1332\ldots < x < 709.7827\ldots$, then:

<6> $-1075 \leq k \leq 1025$ and $|k|$ is an integer on at most 11 bits •

line 63 Properties <4> and <6> give us:

<7> $ln2\_hi \otimes k = ln2\_hi \times k$ and $ln2\_me \otimes k = ln2\_me \times k$ exactly •

By property <5> we have:

$$(x \times inv\_ln2 - 1) \times ln2\_hi \leq k \times ln2\_hi \leq (x \times inv\_ln2 + 1) \times ln2\_hi$$

$$x/2 \leq k \times ln2\_hi \leq 2.x$$

By the Sterbenz theorem (theorem 1, page 5), we have

$$x \ominus (ln2\_hi \otimes k) = x - (ln2\_hi \otimes k)$$

Combined with property <7> we have:

<8> $x \ominus (ln2\_hi \otimes k) = x - (ln2\_hi \times k)$ exactly •

We use conditional Fast2Sum algorithm (with tests on entries), because $x - ln2\_hi \times k$ can be equal to zero (due to the 58 bits of cancellation). The conditional Fast2Sum algorithm (program 2, page 5) leads to

$$r\_hi + u = (x \ominus ln2\_hi \otimes k) + (-ln2\_me \otimes k)$$

With properties <7> and <8> we have:

<9> $r\_hi + u = (x - ln2\_hi \times k) + (-ln2\_me \times k)$ exactly •

line 64 By the property <6> we have:

<10> $|ln2\_lo \times k| \leq 2^{-75}$,
$ln2\_lo \otimes k = (ln2\_lo \times k).(1 + \varepsilon_{-54})$ •

$$\begin{aligned} r\_lo &= u \ominus (ln2\_lo \otimes k) \\ &= (u - (ln2\_lo \otimes k)).(1 + \varepsilon_{-54}) \\ &= (u - (ln2\_lo \times k).(1 + \varepsilon_{-54})).(1 + \varepsilon_{-54}) && \text{<10>} \\ &= (u - (ln2\_lo \times k)).(1 + \varepsilon_{-54}) + \varepsilon_{-129} + \varepsilon_{-183} \end{aligned}$$

That gives us:

<11> $r\_lo = (u - (ln2\_lo \times k)).(1 + \varepsilon_{-54}) + \varepsilon_{-129} + \varepsilon_{-183}$ •

We have:

$$\begin{aligned} r\_hi + r\_lo &= r\_hi + (u - (ln2\_lo \times k)).(1 + \varepsilon_{-54}) + \varepsilon_{-129} + \varepsilon_{-183} && \text{<11>} \\ &= (x - ln2\_hi \times k) + (-ln2\_me \times k) - (ln2\_lo \times k) \\ &\quad + (u - (ln2\_lo \times k)).\varepsilon_{-54} + \varepsilon_{-129} + \varepsilon_{-183} && \text{<9>} \\ &= (x - k.\ln(2)) + k.\varepsilon_{-140} + (u - (ln2\_lo \times k)).\varepsilon_{-54} + \\ &\quad \varepsilon_{-129} + \varepsilon_{-183} && \text{<2>} \\ &= (x - k.\ln(2)) + (u - (ln2\_lo \times k)).\varepsilon_{-54} + \varepsilon_{-128} + \varepsilon_{-183} \end{aligned}$$

In the worst case, we are losing at most 58 bits by cancellation (Table 2). By property <9>, we deduce that $u = 0$ in this case, property <10> ($|ln2\_lo \times k| \leq 2^{-75}$) gives us:

<12> $r\_hi + r\_lo = (x - k \times \ln 2) + \varepsilon_{-127+58=-69}$ •

In addition 69 bits is a precision that can be represented as the sum of 2 floating-point numbers in double precision.

line 66 If $k = 0$ then no subtraction is necessary, then $r\_hi + r\_lo = x$ exactly.

□

At the end of this first rang reduction we have:

$$\exp(x) = 2^k.\exp(r\_hi + r\_lo + \varepsilon_{-69}) = 2^k.\exp(r\_hi + r\_lo).(1 + \varepsilon_{-69})$$

6.2.2 Second range reduction

The number $(r\_hi + r\_lo)$ is still too big to be used in a polynomial evaluation. A second range reduction needs to be done. This second range reduction is based on the additive property of the exponential $e^{a+b} = e^a e^b$, and on the tabulation of some values of the exponential.

Let $index\_flt$ be the ℓ first bits of $(r\_hi + r\_lo)$, then we have:

$$\begin{aligned}\exp(r\_hi + r\_lo) &= \exp(index\_flt).\exp(r\_hi + r\_lo - index\_flt)\\ &\approx (ex\_hi + ex\_lo).\exp(rp\_hi + rp\_lo)\end{aligned}$$

where ex_hi and ex_lo are double precision floating-point numbers extracted from a table addressed by $index\_flt$, such that $ex\_hi + ex\_lo \approx \exp(index\_flt)$. The input argument after this reduction step will be represented as the sum of two double precision floating-point numbers rp_hi and rp_lo such that

$$rp\_hi + rp\_lo = r\_hi + r\_lo - index\_flt$$

Tests show that the optimal table size for the range reduction is 4KBytes [8]. If we want to store these values and keep enough precision (at least 69bits), we need two floating-point numbers (16 bytes) per value.

Let ℓ be the parameter such that $[-2^{-\ell-1}, 2^{-\ell-1}]$ is the range after reduction, we want:

$$\lceil \ln 2.2^{\ell} \rceil 16 \leq (2^{12} = 4096)$$

With $\ell = 8$ we have $\lceil \ln(2).2^8 \rceil 16$ bytes $= 2848$ bytes, and the evaluation range is reduced to $[-2^{-9}, 2^{-9}]$. After this reduction step, we have $|rp\_hi + rp\_lo| \leq 2^{-9}$.
The corresponding sequence of instructions performing this second range reduction is:

Listing 11: Second range reduction

```
/* Constants definition */
static const union{int i[2]; double d;}
#ifdef BIG_ENDIAN
 _two_44_43   = {0x42B80000, 0x00000000}; /*  26388279066624.  */
#else
 ...
#endif
#define two_44_43   _two_44_43.d
#define bias        89;

double ex_hi, ex_lo, index_flt;
int index;

index_flt = (r_hi + two_44_43);
index      = LO(index_flt);
index_flt -= two_44_43;
index     += bias;
r_hi      -= index_flt;

/* Results normalization */
Fast2Sum(rp_hi, rp_lo, r_hi, r_lo)

/* Table lookup */
ex_hi = tab_exp[index][0];
ex_lo = tab_exp[index][1];
```

◇ *Proof.*

line 71 — The constant $two\_44\_43 = 2^{44} + 2^{43}$ is used in rounding to nearest mode to extract the $\ell = 8$ leading bits of $r\_hi + r\_lo$.

line 76 — In the C language, tables have positive indices. We consider positive values as well as negative ones for $index$, therefore we need to use a bias equal to $178/2 = 89$.

line (81, 85) — This sequence of instructions is similar to the one used within DOUBLE2INT (program 6, page 6). It puts in $index$ variable, bits of weight 2^0 to 2^{-8}, minus the value of the bias. Meanwhile, it puts in $index\_flt$ the floating-point value corresponding to the first 8 bits of r_hi.
In line 85 we have:

<13> $r\_hi = r\_hi - index\_flt$ exactly •

line 88 — The Fast2Sum algorithm guarantee:

<14> $|rp\_hi| \leq 2^{-9}$ and $|rp\_lo| \leq 2^{-63}$,
$rp\_hi + rp\_lo = r\_hi + r\_lo$ exactly •

line 91, 92 — We perform table lookup of the 2 values ex_hi and ex_lo. The table is built such that only one cache miss can be encountered in these two table lookups.
By construction of the table tab_exp we have:

<15> $|ex\_hi| \leq 2^{-1}$ and $|ex\_lo| \leq 2^{-55}$
$ex\_hi + ex\_lo = \exp(index\_flt).(1 + \varepsilon_{-109})$ •

□

At the end of this second range reduction we have:

$$\exp(x) = 2^k.(ex\_hi + ex\_lo).\exp(rp\_hi + rp\_lo).(1 + \varepsilon_{-69}).(1 + \varepsilon_{-109})$$

7 Polynomial evaluation

Let $r = (rp\_hi + rp\_lo)$, we need to evaluate $\exp(r)$ with $r \in [-2^{-9}, 2^{-9}]$. We will evaluate $f(r) = (\exp(r) - 1 - r - \frac{r^2}{2})/r^3$ with the following polynom of degree 3:

$$P(r) = c_0 + c_1 r + c_2 r^2 + c_3 r^3$$

where

- $c_0 = 6004799503160629/36028797018963968 \leq 2^{-2}$
- $c_1 = 750599937895079/18014398509481984 \leq 2^{-4}$
- $c_2 = 300240009245077/36028797018963968 \leq 2^{-6}$
- $c_3 = 3202560062254639/2305843009213693952 \leq 2^{-9}$

with c_0, c_1, c_2, c_3 exactly representable by double precision floating-point numbers.
By using `infnorm` function from Maple we get the following error:

<16> $\exp(r) = (1 + r + \frac{1}{2}r^2 + r^3.P(r)).(1 + \varepsilon_{-78})$ with $r \in [-2^{-9}, 2^{-9}]$ •

For efficiency reason, we will evaluate $P(rp\_hi)$ instead of $P(rp\_hi + rp\_lo)$. The error corresponding to this approximation is:

$$\begin{aligned} P(rp\_hi + rp\_lo) - P(rp\_hi) &= c_1.rp\_lo + \\ &\quad c_2.(rp\_lo^2 + 2.rp\_hi.rp\_lo) + \\ &\quad c_3.(rp\_lo^3 + 3.rp\_hi^2.rp\_lo + 3.rp\_hi.rp\_lo^2) \quad <14> \\ &\leq \varepsilon_{-67} + \varepsilon_{-75} \end{aligned}$$

The property <16> becomes:

<17> $\exp(r) = (1 + r + \frac{1}{2}r^2 + r^3.P(rp\_hi)) + \varepsilon_{-78} + \varepsilon_{-86}$ with $r \in [-2^{-9}, 2^{-9}]$ •

$P\_r = P(rp\_hi)$ is evaluated by the following sequences of instructions:

Listing 12: Polynomial evaluation

```
static const union{int i[2]; double d;}
#ifdef BIG_ENDIAN
 _c0          = {0x3FC55555, 0x55555535}, /*   1.6666666666666576923 6e-01 */
 _c1          = {0x3FA55555, 0x55555538}, /*   4.1666666666666463125 7e-02 */
 _c2          = {0x3F811111, 0x31931950}, /*   8.3333427943885873823 e-03 */
 _c3          = {0x3F56C16C, 0x3DC3DC5E}; /*   1.3888890308047167725 1e-03 */
#else
 ...
#endif
#define c0          _c0.d
#define c1          _c1.d
#define c2          _c2.d
#define c3          _c3.d
double P_r;

P_r = (c_0 + rp_hi * (c_1 + rp_hi * (c_2 + (rp_hi * c_3))));
```

◇ *Proof.*

We have:

$\bullet P_0 = c\_3 \otimes rp\_hi$ then $|P_0| \le 2^{-18}$ and $P_0 = (c\_3 \times rp\_hi) + \varepsilon_{-72}$

$\bullet P_1 = c\_2 \oplus P_0$ then $|P_1| \le 2^{-6}$ and $P_1 = (c\_2 + P_0) + \varepsilon_{-60}$

$\bullet P_2 = P_1 \otimes rp\_hi$ then $|P_2| \le 2^{-15}$ and $P_2 = (P_1 \times rp\_hi) + \varepsilon_{-69}$

$\bullet P_3 = c\_1 \oplus P_2$ then $|P_3| \le 2^{-4}$ and $P_3 = (c\_1 + P_2) + \varepsilon_{-58}$

$\bullet P_4 = P_3 \otimes rp\_hi$ then $|P_4| \le 2^{-13}$ and $P_4 = (P_3 \times rp\_hi) + \varepsilon_{-67}$

$\bullet P_5 = c\_0 \oplus P_4$ then $|P_5| \le 2^{-2}$ and $P_5 = (c\_0 + P_4) + \varepsilon_{-56}$

By combining all these errors we get:

<18> $|P\_r| \le 2^{-2}$

$P\_r = (c\_0 + rp\_hi \times (c\_1 + rp\_hi \times (c\_2 + (rp\_hi \times c\_3)))) + \varepsilon_{-55} + \varepsilon_{-65}$

$P\_r = P(rp\_hi) + \varepsilon_{-55} + \varepsilon_{-65}$ •

By the properties <17> and <18>:

<19> $\exp(r) = (1 + r + \frac{1}{2}r^2 + r^3.P\_r) + \varepsilon_{-78} + \varepsilon_{-81}$ with $r \in [-2^{-9}, 2^{-9}]$ •

□

At the end of the polynomial evaluation scheme we have:

$$\exp(x) = 2^k.(ex\_hi + ex\_lo).(1 + r + \frac{1}{2}r^2 + r^3.P\_r + \varepsilon_{-78} + \varepsilon_{-81}).(1 + \varepsilon_{-69}).(1 + \varepsilon_{-109})$$

8 Reconstruction

Along previous step of the algorithm we get the following results:

- k, r_hi and r_lo during the additive range reduction,
- ex_hi, ex_lo, rp_hi and rp_lo during the table range reduction,
- P_r with the polynomial range reduction.

The reconstruction step consist in merging all these results in order to get $\exp(x)$. This step is based on the following mathematical formula:

$$\exp(x) = 2^k.(ex\_hi + ex\_lo).(1 + (rp\_hi + rp\_lo) + \frac{1}{2}.(rp\_hi + rp\_lo)^2 + (rp\_hi + rp\_lo)^3.P\_r)$$

However, some terms in this equation are too small compared to dominants terms, and should not be taken into account: We approximate:

$$Rec = (ex\_hi + ex\_lo).(1 + (rp\_hi + rp\_lo) + \frac{1}{2}.(rp\_hi + rp\_lo)^2 + (rp\_hi + rp\_lo)^3.P\_r)$$

by

$$Rec^* = ex\_hi \times (1 + rp\_hi + rp\_lo + \frac{1}{2}(rp\_hi)^2 + P\_r \times (rp\_hi)^3) + ex\_lo \times (1 + rp\_hi + \frac{1}{2}(rp\_hi)^2)$$

The corresponding error is given by:

$$\begin{array}{l} Rec - Rec^* = \\ (ex\_hi + ex\_lo).(rp\_hi.rp\_lo + \frac{1}{2}rp\_lo^2) + \\ ex\_hi.rp\_lo.(3.rp\_hi^2 + 3.rp\_hi.rp\_lo + rp\_lo^2) + \\ ex\_lo.rp\_hi.(3.rp\_lo^2 + 3.rp\_hi.rp\_lo + rp\_hi^2) + ex\_lo.rp\_lo^3 + \\ ex\_lo.rp\_lo \\ \leq 2^{-74} + 2^{-82} + 2^{-88} \end{array}$$

Hence the following property:

<20> The error done when approximating Rec by Rec^* is:

$$Rec = Rec^* + \varepsilon_{-74} + \varepsilon_{-81}$$

•

The order in which are executed the instructions is choosen in order to minimize the error. These terms and the intermediate computations with their order of magnitude are given in Figure 1, page 21.

Listing 13: Reconstruction

```
double R1, R2, R3_hi, R3_lo, R4, R5_hi, R5_lo, R6, R7, R8, R9, R10, R11, crp_hi;


R1 = rp_hi * rp_hi;

crp_hi = R1 * rp_hi;
/* Correspond to R1 /= 2; */
HI(R1) = HI(R1)-0x00100000;

R2 =  P_r * crp_hi;

Dekker(R3_hi, R3_lo, ex_hi, rp_hi);
R4 = ex_hi * rp_lo;

Dekker(R5_hi, R5_lo, ex_hi, R1);
R6 = R4 + (ex_lo * (R1 + rp_hi));

R7  = ex_hi * R2;
R7 += (R6 + R5_lo) + (R3_lo + ex_lo);

Fast2Sum(R9, R8, R7, R5_hi);

Fast2Sum(R10, tmp, R3_hi, R9);
R8 += tmp;

Fast2Sum(R11, tmp, ex_hi, R10);
R8 += tmp;

Fast2Sum(R11, R8, R11, R8);
```

◇ *Proof.*

line 112 $|rp\_hi| \leq 2^{-9}$, therefore:

<21> $|R1| \leq 2^{-18}$,
$R1 = (rp\_hi)^2.(1+\varepsilon_{-54})$ •

line 114 By using property <21> and $|rp\_hi| \leq 2^{-9}$ we get:

<22> $|crp\_hi| \leq 2^{-27}$,
$crp\_hi = (rp\_hi)^3.(1+\varepsilon_{-53})$ •

line 116 This operation is a division by 2, done by subtracting 1 to the exposant. This oepration is valid and exact if $R1$ is not a denormal number, which is the case (property <1> $|x| \geq 2^{-54}$), and the table 2 showing that we have at most 58 bits of cancellation).

<23> $|R1| \leq 2^{-19}$,
$R1 = \frac{1}{2}(rp\_hi)^2.(1+\varepsilon_{-54})$ •

line 118 By using properties <18> and <22>:

<24> $|R2| \leq 2^{-29}$,
$R2 = P\_r \times (crp\_hi).(1+\varepsilon_{-54})$,
$R2 = P\_r \times (rp\_hi)^3.(1+\varepsilon_{-52})$ •

line 120 By using Dekker algorithm (programme 4, page 6) and properties <14> and <15> we have:

<25> $|R3\_hi| \leq 2^{-10}$ and $|R3\_lo| \leq 2^{-64}$,
$R3\_hi + R3\_lo = ex\_hi \times rp\_hi$ exactly •

line 121 By using properties <14> and <15>:

<26> $|R4| \leq 2^{-64}$,
$R4 = ex\_hi \times rp\_lo.(1+\varepsilon_{-54})$ •

line 123 By using Dekker algorithm and properties <15> and <23> we have:

<27> $|R5\_hi| \leq 2^{-20}$ and $|R5\_lo| \leq 2^{-74}$,
$(R5\_hi + R5\_lo) = ex\_hi \times R1$ exactly,
$(R5\_hi + R5\_lo) = (ex\_hi \times \frac{1}{2}(rp\_hi)^2).(1+\varepsilon_{-54})$ •

line 124 By using properties <15> and <23>:
$|R1 + rp\_hi| \leq 2^{-8}$,
$R1 \oplus rp\_hi = R1 + rp\_hi + \varepsilon_{-62}$,
$|ex\_lo \times (R1 + rp\_hi)| \leq 2^{-63}$,
$ex\_lo \otimes (R1 \oplus rp\_hi) = ex\_lo \times (\frac{1}{2}(rp\_hi)^2 + rp\_hi) + \varepsilon_{-116}$.
which combined with property <26> gives us:

<28> $|R6| \leq 2^{-62}$,
$R6 = R4 + \big(ex\_lo \times \big(\frac{1}{2}(rp\_hi)^2 + rp\_hi\big) + \varepsilon_{-116}\big) + \varepsilon_{-116}$
$R6 = (ex\_hi \times rp\_lo) + \big(ex\_lo \times \big(\frac{1}{2}(rp\_hi)^2 + rp\_hi\big)\big) + \varepsilon_{-115} + \varepsilon_{-118}$ •

line 126 By using properties <15> and <24> we get:

<29> $|R7| \leq 2^{-30}$,
$R7 = (ex\_hi \times R2).(1 + \varepsilon_{-54})$,
$R7 = (ex\_hi \times P\_r \times (rp\_hi)^3).(1 + \varepsilon_{-52} + \varepsilon_{-53})$ •

line 127 By using properties <27> and <28> we get:

<30> $|R6 + R5\_lo| \leq 2^{-61}$,
$R6 \oplus R5\_lo = R6 + R5\_lo + \varepsilon_{-115}$ ou
$R6 \oplus R5\_lo = (ex\_hi \times rp\_lo) + (ex\_lo \times (\frac{1}{2}(rp\_hi)^2 + rp\_hi)) + R5\_lo + \varepsilon_{-113}$ •

By using properties <15> and <25> we get:

<31> $|R3\_lo + ex\_lo| \leq 2^{-54}$,
$R3\_lo \oplus ex\_lo = R3\_lo + ex\_lo + \varepsilon_{-108}$ •

By using properties <30> and <31> we get:
$|R6 + R5\_lo + R3\_lo + ex\_lo| \leq 2^{-53}$ et
$(R6 \oplus R5\_lo) \oplus (R3\_lo \oplus ex\_lo) = (R6 \oplus R5\_lo) + (R3\_lo \oplus ex\_lo) + \varepsilon_{-107}$.
which combined with property <29> gives us:

<32> $|R7| \leq 2^{-29}$,

$$\begin{aligned} R7 &= (ex\_hi \times P\_r \times (rp\_hi)^3).(1 + \varepsilon_{-52} + \varepsilon_{-53}) + \\ &\quad ((ex\_hi \times rp\_lo) + (ex\_lo \times (\tfrac{1}{2}(rp\_hi)^2 + rp\_hi)) + R5\_lo + \varepsilon_{-113}) + \\ &\quad (R3\_lo + ex\_lo + \varepsilon_{-108}) \\ &= ex\_hi \times (rp\_lo + P\_r \times (rp\_hi)^3) + \\ &\quad ex\_lo \times (1 + rp\_hi + \tfrac{1}{2}(rp\_hi)^2) + \\ &\quad R5\_lo + R3\_lo + \varepsilon_{-80} \end{aligned} \quad \bullet$$

line 129 Fast2Sum algorithm guarantees:

<33> $|R9| \leq 2^{-19}$ and $|R8| \leq 2^{-73}$,
$R7 + R5\_hi = R9 + R8$ exactly •

line 131 Fast2Sum algorithm guarantee:

<34> $|R10| \leq 2^{-9}$ and $|tmp| \leq 2^{-63}$,
$R3\_hi + R9 = R10 + tmp$ exactly •

line 132 By using properties <33> and <34>:

<35> $|R8 + tmp| \leq 2^{-62}$,
$R8 = R8 + tmp + \varepsilon_{-116}$ •

line 134 Fast2Sum algorithm guarantees:

<36> $|R11| \leq 2^0$ and $|tmp| \leq 2^{-54}$,
$R11 + tmp = ex\_hi + R10$ exactly •

line 135 By using properties <35> and <36>:

<37> $|R8 + tmp| \leq 2^{-53}$,
$R8 = R8 + tmp + \varepsilon_{-107}$ •

line 137 Fast2Sum algorithm guarantees:

<38> $|R11| \leq 2^1$ and $|R8| \leq 2^{-53}$,
$R11 + R8 = R11 + R8$ exactly •

□

Therefore we have:

$$
\begin{aligned}
R11 + R8 &= R11 + tmp + R8 + \varepsilon_{-107} && \text{<37>}\\
&= ex\_hi + R10 + R8 + \varepsilon_{-107} && \text{<36>}\\
&= ex\_hi + R10 + tmp + R8 + \varepsilon_{-107} + \varepsilon_{-116} && \text{<35>}\\
&= ex\_hi + R3\_hi + R9 + R8 + \varepsilon_{-106} && \text{<34>}\\
&= ex\_hi + R3\_hi + R7 + R5\_hi + \varepsilon_{-106} && \text{<33>}\\
&= ex\_hi + R3\_hi + R5\_hi + \\
&\quad ex\_hi \times (rp\_lo + P\_r \times (rp\_hi)^3) + \\
&\quad ex\_lo \times (1 + rp\_hi + \tfrac{1}{2}(rp\_hi)^2) + \\
&\quad R5\_lo + R3\_lo + \varepsilon_{-80} + \varepsilon_{-106} && \text{<32>}\\
&= (R3\_hi + R3\_lo) + (R5\_hi + R5\_lo) + \\
&\quad ex\_hi \times (1 + rp\_lo + P\_r \times (rp\_hi)^3) + \\
&\quad ex\_lo \times (1 + rp\_hi + \tfrac{1}{2}(rp\_hi)^2) + \varepsilon_{-80} + \varepsilon_{-106}\\
&= (R3\_hi + R3\_lo) + \\
&\quad ex\_hi \times (1 + rp\_lo + \tfrac{1}{2}(rp\_hi)^2 + P\_r \times (rp\_hi)^3) + \\
&\quad ex\_lo \times (1 + rp\_hi + \tfrac{1}{2}(rp\_hi)^2) + \varepsilon_{-74} + \varepsilon_{-79} && \text{<27>}\\
&= ex\_hi \times (1 + rp\_hi + rp\_lo + \tfrac{1}{2}(rp\_hi)^2 + P\_r \times (rp\_hi)^3) + \\
&\quad ex\_lo \times (1 + rp\_hi + \tfrac{1}{2}(rp\_hi)^2) + \\
&\quad \varepsilon_{-74} + \varepsilon_{-79} && \text{<25>}
\end{aligned}
$$

Using property <20> we get:

$$
\begin{aligned}
R11 + R8 &= (ex\_hi + ex\_lo).(1 + r + \tfrac{1}{2}.r^2 + r^3.P\_r)\\
&\quad + \varepsilon_{-73} + \varepsilon_{-78}
\end{aligned}
$$

By construction of values $(ex\_hi + ex\_lo)$, we have $(ex\_hi + ex\_lo) > \exp(-1) > 2^{-2}$, therefore

<39> $|R11 + R8| > 2^{-2}$

and

$\exp(x) = 2^k.(R11 + R8 + \varepsilon_{-73} + \varepsilon_{-78}).(1 + \varepsilon_{-69}).(1 + \varepsilon_{-109})$ •

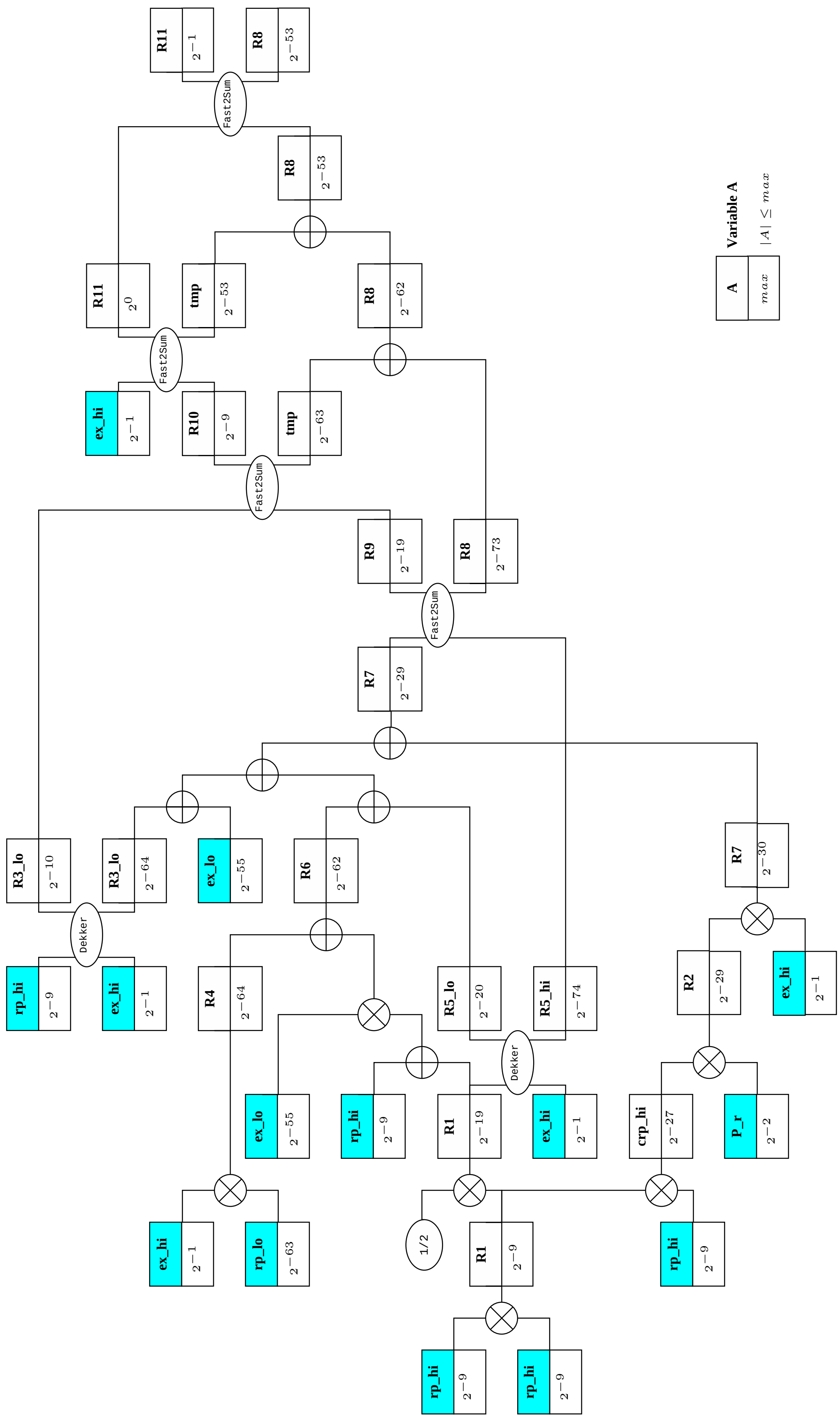

Figure 1: The reconstruction step

9 Test if correct rounding is possible

We now have to round correctly the result and multiply it by 2^k, with k an integer. This multiplication is exact, therefore we have:

$$\begin{aligned}\exp(x) &= \circ\left(2^k.(R11 + R8 + \varepsilon_{-73} + \varepsilon_{-78}).(1+\varepsilon_{-69}).(1+\varepsilon_{-109})\right)\\ &= 2^k.\circ\left((R11 + R8 + \varepsilon_{-73} + \varepsilon_{-78}).(1+\varepsilon_{-69}).(1+\varepsilon_{-109})\right)\end{aligned}$$

if the result is not a denormal number.

However, if the final result belongs to denormal numbers, then the precision of the result is less than the 53 bits of a normal number. Let us take an exemple. Let $a = 1.75$ be a floating-point number exactly representable in double precision format (we have $\circ(a) = a$). Let us multiply this number by 2^{-1074}. The exact result is 1.75×2^{-1074} which is different from the result rounded to nearest in double precision $\circ(1.75 \times 2^{-1074}) = 2^{-1073}$.

Therefore, in the case when the result is a denormal number, we have to use a special procedure.

9.1 Rounding to nearest

Listing 14: Test if rounding to nearest is possible

```
static const union{int i[2]; double d;}
#ifdef BIG_ENDIAN
 _two1000    = {0x7E700000, 0x00000000}, /*  1.07150860718626732095e301  */
 _twom1000   = {0x01700000, 0x00000000}, /*  9.33263618503218878990e-302 */
 _errn       = {0x3FF00080, 0x00000000}, /*  1.00012207031250000000e0    */
 _twom75     = {0x3B400000, 0x00000000}; /*  2.64697796016968855958e-23  */
#else
 ...
#endif
#define two1000    _two1000.d
#define twom1000   _twom1000.d
#define errn       _errn.d
#define twom75     _twom75.d

int errd        = 71303168;                  /* 68 * 2^20 */

double R11_new, R11_err, R13, st2mem;
int    exp_R11;

union { int i[2]; long long int l; double d;} R12;


/* Résult = (R11 + R8) */
if (R11 == (R11 + R8 * errn)){
  if (k > -1020){
    if (k < 1020){
      HI(R11) += (k<<20);
      return R11;
    }else {
      /* we are close to  + Inf */
      HI(R11) += ((k-1000)<<20);
      return R11*two1000;
    }
  }else {
    /* We consider denormal number */
    HI(R11_new) = HI(R11)+((k+1000)<<20);
    LO(R11_new) = LO(R11);
    R12.d       = R11_new * twom1000;

    HI(st2mem) = R12.i[HI_ENDIAN];
    LO(st2mem) = R12.i[LO_ENDIAN];

    R11_err -= st2mem * two1000;
    HI(R13) = HI(R11_err) & 0x7fffffff;
    LO(R13) = LO(R11_err);

    if (R13 == two_m75){
      exp_R11 = (HI(R11) & 0x7ff00000) - errd;
      if ((HI(R8) & 0x7ff00000) < exp_R11){
        /* Difficult rounding ! */
        sn_exp(x);
```

```
189        }
190        /* The error term is exactly 1/2 ulp */
191        if ((HI(R11_err) > 0) && (HI(R8) > 0)) R12.l +=1;
192        else
193        if ((HI(R11_err) < 0) && (HI(R8) < 0)) R12.l -=1;
194      }
195      return R12.d;
196    }
197  } else {
198    /* Challenging case */
199    sn_exp(x);
200  }
```

◇ *Proof.*

line 161 — This test is used to know whether we can round the result or not. More details about this trick can be found in [9].
By using property <39> we have:

$$|(R11 + R8) \times 2^k - \exp(x)| \leq 2^{-68}$$

where $x \in [A, B]$, that leads to $errn = 1 + 2^{-68} \times 2^{55} = 1 + 2^{-13}$. This test is true if we are able to round correctly the result, else we need to call multiprecision procedure.

line 162-170 — We are able to round, now we need to perform the multiplication $R11 \times 2^k$ exactly. We do this multiplication by using integer addition on the exponent of $R11$. For this operation to be valid and exact, we must be sure not to create a denormal or infinity. This is the reason why we perform a test on the value of k.
$(R11 + R8) > 2^{-2}$ then $2^k.(R11 + R8)$ will not lead to a denormal number if $k > -1020$.
$(R11 + R8) < 2^3$ then $2^k.(R11 + R8)$ will not lead to an overflow if $k < 1020$. Then we have $R11 = R11 \oplus R8$
In the case when we may return an overflow as result, we make the value of k smaller, by subtracting 1000 to it. This will prevent the apparition of exception cases during the addition of k to the exponent. The result is then multiplied by the floating-point number $twom1000 = 2^{-1000}$. This multiplication is exact but in case of underflow, in which case exceptions will be properly raised.

line 173 — The result may be a denormal number. We need to use a specific test to check whether if we are able to round properly the result.

$$R11 = R11 \times 2^{k+1000}$$

line 175 — In rounding to nearest we get:

$$R12 = R11 \otimes 2^{-1000} = R11 \times 2^{-1000} + \varepsilon_{-1075}$$

The error term ε_{-1075} comes from the possible truncation when the result is a denormal number.

line 177,178 — Processors do not handle denormal number in 'hardware', there are treated when they are stored in memory. It means that number within register can't be denormal. Therefore, these lines prevent the compiler from performing "dangerous" optimizations, meanwhile preventing to have extra precision by forcing $R12$ to transit through memory. These lines could be removed for exemple by using gcc and the flag `-ffloat-store` that will have the same effect. However this flag forces each floating-point instruction to transit through memory, and has as consequence to severely degrade the performance of the resulting program. The solution to keep good performance is to manually force a data to transit through memory, in order to have an IEEE compliant behavior for denormal numbers.

line 180 Let

$$R11\_err = R11 \ominus R12 \otimes 2^{1000}$$

By the Sterbenz lemma, and by the fact that a multiplication by a power of 2 is exact we have:

$$R11\_err = R11 - R12 \times 2^{1000}$$

This operation will put within $R11\_err$ the error done during the multiplication of $R11$ by 2^{-1000} (line 175).

line 181,182 Remove the sign information of $R11\_err$

$$R13 = |R11\_err|$$

line 184 Test if $R13$ is exactly equal to the absolute error (which is also the relative error) done during the rounding process, in rounding to nearest, whithin denormal number, times 2^{1000}:

$$2^{-1075} \times 2^{1000} = 2^{-75}$$

Now we want to prove that if error term is strictly less than $1/2ulp(R12)$ then $R12$ corresponds to the correct rounding of $R11 + R8$.
If $|R11\_err| < \frac{1}{2}ulp(R12)$ then
$|R11\_err| \leq \frac{1}{2}ulp(R12) - ulp(R11)$ and $|R8| \leq \frac{1}{2}ulp(R11)$ therefore:

$$|R11\_err + R8| \leq \frac{1}{2}ulp(R12) - ulp(R11) + \frac{1}{2}ulp(R11)$$

then:

$$|R11\_err + R8| < \frac{1}{2}ulp(R12)$$

In that case $R12$ represents the correct rounding of $R11 + R8$ si $|R11\_err| < \frac{1}{2}ulp(R12)$.
However, if $|R11\_err| = \frac{1}{2}ulp(R12) = 2^{-75}$, during the multiplication $R11 \times 2^{-1000}$, the result is rounded to odd/even due to the presence of an ambiguous value. It mean that $R12$ may not represent the rounding to nearest result of $R11 + R8$, we need to perform a correction:

- If $R8 > 0$, and $R11\_err = -\frac{1}{2}ulp(R12)$, then the round to odd/even was done on the correct side.
- If $R8 > 0$, and $R11\_err = \frac{1}{2}ulp(R12)$, then the round to odd/even wasn't done on the correct side. We need to perform a correction by adding $1ulp$ to $R12$ (figure 2, case a)
- If $R8 < 0$, and $R11\_err = -\frac{1}{2}ulp(R12)$, then the round to odd/even wasn't done on the correct side. We need to perform a correction by subtracting $1ulp$ to $R12$ (figure 2, case b).
- If $R8 < 0$, and $R11\_err = \frac{1}{2}ulp(R12)$, then the round to odd/even was done on the correct side.

line 186 When we are in presence of a consecutive sequence of 0 or 1 straddling $R11$ and $R8$, then the test done at line 161 will not detect a possible problem. This problem will arise only with denormal numbers, when $R11\_err$ is close to $\frac{1}{2}ulp(R12)$.
Therefore we have to detect if in that case (denormal, $|R11\_err| = \frac{1}{2}ulp(R12)$) we have enough precision to correctly round the result. We use a test similar to the one used to test wheteher we can round with rounding toward $\pm\infty$. Indeed, problematic cases arise when $\frac{1}{2}ulp(R12) - 2^{-68}.R11 \leq |R11\_err + R8| \leq \frac{1}{2}ulp(R12) + 2^{-68}.R11)$.

line 191,193 Test if we are in presence of one of the cases described previously, and correct the result by adding or subtracting $1ulp$ by using the "continuity" of the representation of floating-point number.

□

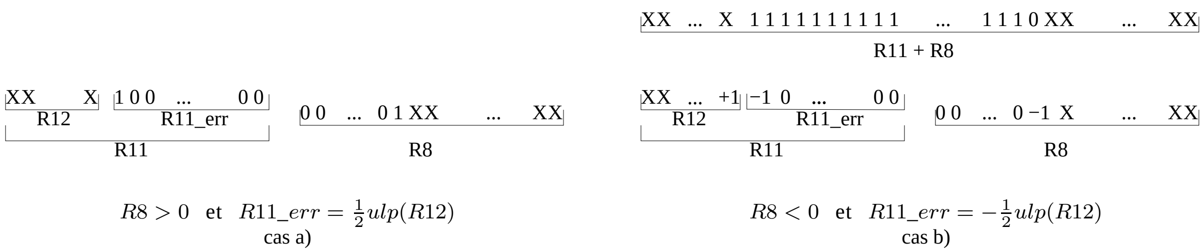

Figure 2: Description of problem with rounding to nearest of a denormal number.

9.2 Rounding toward $+\infty$

Listing 15: Test if rounding toward $+\infty$ is possible

```
static const union{int i[2]; double d;}
#ifdef BIG_ENDIAN
 _two1000      = {0x7E700000, 0x00000000}, /*  1.07150860718626732095e301   */
 _twom1000     = {0x01700000, 0x00000000}, /*  9.33263618503218878990e-302 */
#else
 ...
#endif
#define two1000     _two1000.d
#define twom1000    _twom1000.d


int errd          = 71303168;                  /* 68 * 2^20 */

int    exp_R11;
union {int i[2]; long long int l; double d;} R12;

/* Result = (R11 + R8) */

exp_R11 = (HI(R11) & 0x7ff00000) - errd;

if ((HI(R8) & 0x7ff00000) > exp_R11){
  /* We are able to round the result */
  if (k > -1020){
    if (k < 1020){
      HI(R11) += (k<<20);
    }else {
      /* We are close to + Inf */
      HI(R11) += ((k-1000)<<20);
      R11 *= two1000;
    }
    if (HI(R8) > 0){
      R12.d  = R11;
      R12.l += 1;
      R11    = R12.d;
    }
    return R11;
  }else {
    /* We are with denormal number */
    HI(R11) += ((k+1000)<<20);
    R12.d    = R11 * twom1000;

    HI(st2mem) = R12.i[HI_ENDIAN] ;
    LO(st2mem) = R12.i[LO_ENDIAN]

    R11 -= st2mem * two1000;
```

```
246       if ((HI(R11) > 0)||((HI(R11) == 0)&&(HI(R8) > 0))) R12.l += 1;
247
248       return R12.d;
249     }
250   }else {
251     /* Difficult case */
252     su_exp(x);
253   }
```

◇ *Proof.*

The program used to check whether correct rounding toward $+\infty$ is possible is similar to the one used with rounding to nearest.

line 221	This test is valid even if the final result is a denormal number.
line 231	We add $1ulp$ to the result if $R8$ is positive.
line 245	Like for rounding to nearest, the quantity $R11$ represents the rounding error that comes from the operation $R11 * twom1000$ in line 240.
line 246	This test checks whether the error from line 240 is strictly positive or if it is equal to zero and if $R8$ is strictly positive. If we are in one of these two cases, by definition of rounding toward $+\infty$, we need to add $1ulp$ to the result.

□

9.3 Rounding toward $-\infty$

Listing 16: Test if rounding toward $-\infty$ is possible

```
static const union{int i[2]; double d;}
#ifdef BIG_ENDIAN
 _two1000    = {0x7E700000, 0x00000000}, /*  1.07150860718626732095e301   */
 _twom1000   = {0x01700000, 0x00000000}, /*  9.33263618503218878990e-302 */
#else
 ...
#endif
#define two1000    _two1000.d
#define twom1000   _twom1000.d

int errd        = 71303168;                /* 68 * 2^20 */

int    exp_R11;
union {int i[2]; long long int l; double d} R12;

/* Résult = (R11 + R8) */

exp_R11 = (HI(R11) & 0x7ff00000) - errd;

if ((HI(R8) & 0x7ff00000) > exp_R11){
  /* We are able to round the result */
  if (k > -1020){
    if (k < 1020){
      HI(R11) += (k<<20);
    }else {
      /* We are close to + Inf */
      HI(R11) += ((k-1000)<<20);
      R11 *= two1000;
    }
    if (HI(R8) > 0){
      R12.d  = R11;
      R12.l += 1;
      R11    = R12.d;
    }
    return R11;
  }else {
    /* We are with denormal number */
    HI(R11) += ((k+1000)<<20);
    R12.d    = R11 * twom1000;

    HI(st2mem) = R12.i[HI_ENDIAN];
    LO(st2mem) = R12.i[LO_ENDIAN];
```

```

    R11 -= st2mem * two1000;
    if ((HI(R11) < 0)||((HI(R11) == 0)&&(HI(R8) < 0))) R12.l -= 1;

    return R12.d;
  }
}else {
  /* Difficult case */
  su_exp(x);
}
```

◇ *Proof.*

The program used to check whether correct rounding toward $-\infty$ is possible is similar to the previous one.

□

9.4 Rounding toward 0

The program used to check whether correct rounding toward 0 is possible is identical to the one used for rounding to $-\infty$ because $\exp(x)$ is a positive function.

10 Accurate phase

When the previous computation failed, it means that the rounding of the result is difficult to decide. We need to use more accurate methods:

- sn_exp with rounding to nearest,
- su_exp with rounding toward $+\infty$,
- sd_exp with rounding toward $-\infty$,

These methods are based on SCS library [5], with 30 bits of precision per digit and 8 digits per vector. The guaranteed precision with this format is 211 bits at least. Even if there is no proof for these operators yet, the proof for correct rounding of the exponential only requires the following properties, which are easy to check and / or satisfy:

Property 2 *(Addition) Let $a \boxplus b$ represent the multiprecision operation performing an addition between a and b with at least 210 bits of precision for the result. Like for double precision floating point number, the SCS addition may lead to a cancellation. We have:*

$$a + b = (a \boxplus b).(1 + \varepsilon_{-211})$$

Property 3 *(Multiplication) Let $a \boxtimes b$ represent the multiprecision operation performing a multiplication between a and b with at least 210 bits of precision for the result. This operation does not produce a cancellation.*

$$a \times b = (a \boxtimes b).(1 + \varepsilon_{-211})$$

10.1 Overview of the algorithm

Here is the algorithm used for the second part of the evaluation:

1. **No special case handling**
   Special caseshave been handled by the first part.

2. **Range reduction**
   We compute the reduced argument r and the integer k such that:

$$r = \frac{x - k.\ln(2)}{512}$$

with $\frac{-\ln(2)}{1024} \leq r \leq \frac{-\ln(2)}{1024}$
such that

$$\exp(x) = \exp(r)^{512} \times 2^k$$

3. **Polynomial evaluation**
We compute the polynom $P(r)$ of degree 11:

$$\exp(r) = (1 + r + P(r)).(1 + \varepsilon_{-179})$$

4. **Powering the result**

$$\exp(r)^{512} = \left(\left(\left(\left(\left(\left(\left(\left(\exp(r)^2\right)^2\right)^2\right)^2\right)^2\right)^2\right)^2\right)^2\right)^2$$

5. **Reconstruction**

$$\exp(x) = \exp(r)^{512}.2^k.(1 + \varepsilon)$$

with $|\varepsilon| \leq 2^{-170}$

We have choosen this evaluation scheme, because the reconstruction step use the squaring multiprecision operator. This operator facilitate the error computation and is very economic: its cost is 0.7 times that of a true multiprecision multiplication.

We will notice that there exists an alternative to the squaring solution. We can tabulate values $2^{\frac{N}{512}}$ for $N = 1, 2, \ldots, 511$ and use the formula $\exp(x) = \exp(r) \times 2^N \times 2^{\frac{M}{512}}$ with $k = M + N/512$. However we prefer the squaring method that do not request the storage of SCS numbers and the associated quantity of memory.

10.2 Function calls

With Lefèvre worst cases on table makers dilemma we get the following theorem:

Theorem 6 *(Correct rounding for the exponential) Let y be the exact value of the exponential of a floating-point number in double precision x. Let y^* be an approximation of y such that the distance between y and y^* be bounded by ε. Then if $\varepsilon \leq 2^{-157}$, for each of the four rounding mode, rounding y^* is equivalent to rounding y;*

To round the multiprecision result in SCS format depending on the rounding mode, we use the following procedure (`scs_get_d`, `scs_get_d_pinf`, `scs_get_d_minf`).

10.2.1 Rounding to nearest

Listing 17: Compute the rounding to nearest of the exponential in multiprecision

```
double sn_exp(double x){
  scs_t res_scs;
  scs_db_number res;

  exp_SC(res_scs, x);
  scs_get_d(&res.d, res_scs);  res.d = x;

  return res.d;
}
```

10.2.2 Rounding toward $+\infty$

Listing 18: Compute the rounding toward $+\infty$ of the exponential in multiprecision

```
double su_exp(double x){
  scs_t res_scs;
  scs_db_number res;

  exp_SC(res_scs, x);
  scs_get_d_pinf(&res.d, res_scs);
  return res.d;
}
```

10.2.3 Rounding toward $-\infty$

Listing 19: Compute the rounding toward $-\infty$ of the exponential in multiprecision

```
double sd_exp(double x){
  scs_t res_scs;
  scs_db_number res;

  exp_SC(res_scs, x);
  scs_get_d_minf(&res.d, res_scs);
  return res.d;
}
```

10.3 Software

The function exp_SC approximate the exponential of x with 170 bits of precision and put the result in res_scs.

Listing 20: Compute the exponential in multiprecision

```
void exp_SC(scs_ptr res_scs, double x){
  scs_t sc1, red;
  scs_db_number db;
  int i, k;


  /* db.d = x/512  (= 2^9)  */

  db.d = x;
  db.i[HI_ENDIAN] -= (9 << 20);
  scs_set_d(sc1, db.d);


  DOUBLE2INT(k, (db.d * iln2_o512.d));

  /* 1) Range reduction */

  scs_set(red,     sc_ln2_o512_ptr_1);;
  scs_set(red_low, sc_ln2_o512_ptr_2);
  if (k>0){
    scs_mul_ui(red,     (unsigned int) k);
    scs_mul_ui(red_low, (unsigned int) k);
  }else {
    scs_mul_ui(red,     (unsigned int)(-k));
    scs_mul_ui(red_low, (unsigned int)(-k));
    red->sign *= -1;
    red_low->sign *=-1;
  }

  scs_sub(red, sc1, red);
  scs_sub(red, red, red_low);


  /* 2) Polynomial evaluation */

  scs_mul(res_scs, constant_poly_ptr[0], red);
  for(i=1; i< 11; i++){
    scs_add(res_scs, constant_poly_ptr[i], res_scs);
    scs_mul(res_scs, red, res_scs);
```

```
40    }
41
42    scs_add(res_scs, SCS_ONE, res_scs);
43    scs_mul(res_scs, red, res_scs);
44    scs_add(res_scs, SCS_ONE, res_scs);
45
46    /* 3) Powering the result exp(r)^512 */
47
48    for(i=0; i<9; i++){
49      scs_square(res_scs, res_scs);
50    }
51
52    /* 4) Multiplication by 2^k */
53
54    res_scs->index += (int)(k/30);
55    if ((k%30) > 0)
56      scs_mul_ui(res_scs, (unsigned int)(1<<((k%30))));
57    else if ((k%30) < 0){
58      res_scs->index --;
59      scs_mul_ui(res_scs, (unsigned int)(1<<((30+(k%30)))));
60    }
61
62  }
```

◇ *Proof.*

line 9 — $db.d = x$

line 10 — This operation divides $db.d$ by $512 = 2^9$ and is valid under the condition that $db.d$, and consequently x, do not represent a special values (denormal, infinity, NaN). This condition is satisfied because special cases have been treated during the quick phase.

line 11 — $sc1$ is a 211 bits multiprecision number such that:

<40> $sc1 = db.d = \frac{x}{512}$ exactly •

line 14 — $iln2\_o512.d$ is a double precision floating-point number such that: $iln2\_o512.d = \frac{512}{\ln 2}(1+\varepsilon_{-54})$. This line puts in k the integer closest to $db.d \otimes \frac{512}{\ln 2}$. We use the property of $DOUBLE2INT$ which converts a floating-point number into an integer with rounding to nearest.
Moreover k satisfies the following property:

<41>

$$\lfloor \frac{x}{\ln 2} \rfloor \le k \le \lceil \frac{x}{\ln 2} \rceil \quad \text{et} \quad -1075 \le |k| \le 1025$$

•

And k is a 11 bits integer.

line 18, 19 — By construction we have:

$$red + red\_low = \frac{\ln 2}{512}(1+\varepsilon_{-450})$$

and red is constructed in order to make the multiplication of red by k exact if $|k| \le 2^{11}$.

line 28 — At the end of the test on k we have: $red + red\_low = k \boxtimes \frac{\ln 2}{512}(1+\varepsilon_{-450})$ with $|k| < 2^{11}$, then:

<42>

$$red + red\_low = k \times \frac{\ln 2}{512}(1+\varepsilon_{-411})$$

•

line 30,31 By the properties <40> and <42> we have:

<43> $red = \frac{x}{512} \boxminus \left(k \times \frac{\ln 2}{512}\right)(1 + \varepsilon_{-411})$ •

In addition we have seen in the quick phase that at most 58 bits could be cancelled during this subtraction.

<44> $|red| \leq \frac{\ln 2}{1024} \leq 2^{-10}$,
$red = \frac{x}{512} - k \times \frac{\ln 2}{512} + \varepsilon_{-210}$ •

line 34-44 We now perform the polynomial evaluation where the coefficient have the following properties.

<45> $|constant\_poly\_ptr[0] = c_0| \leq 2^{-28}$, $|constant\_poly\_ptr[1] = c_1| \leq 2^{-25}$, $|constant\_poly\_ptr[2] = c_2| \leq 2^{-21}$, $|constant\_poly\_ptr[3] = c_3| \leq 2^{-18}$, $|constant\_poly\_ptr[4] = c_4| \leq 2^{-15}$, $|constant\_poly\_ptr[5] = c_5| \leq 2^{-12}$, $|constant\_poly\_ptr[6] = c_6| \leq 2^{-9}$, $|constant\_poly\_ptr[7] = c_7| \leq 2^{-6}$, $|constant\_poly\_ptr[8] = c_8| \leq 2^{-4}$, $|constant\_poly\_ptr[9] = c_9| \leq 2^{-2}$, $|constant\_poly\_ptr[10] = c_{10}| \leq 2^{-1}$ •

We have:

- $P_0 = c_1 \boxplus (red \boxtimes c_0)$ therefore $|P_0| \leq 2^{-24}$ et $P_0 = (c_1 + (red \times c_0))(1 + \varepsilon_{-210} + \varepsilon_{-244})$
- $P_1 = c_2 \boxplus (red \boxtimes P_0)$ therefore $|P_1| \leq 2^{-20}$ et $P_1 = (c_2 + (red \times P_0))(1 + \varepsilon_{-210} + \varepsilon_{-240})$
- $P_2 = c_3 \boxplus (red \boxtimes P_1)$ therefore $|P_2| \leq 2^{-17}$ et $P_2 = (c_3 + (red \times P_1))(1 + \varepsilon_{-210} + \varepsilon_{-236})$
- $P_3 = c_4 \boxplus (red \boxtimes P_2)$ therefore $|P_3| \leq 2^{-14}$ et $P_3 = (c_4 + (red \times P_2))(1 + \varepsilon_{-210} + \varepsilon_{-233})$
- $P_4 = c_5 \boxplus (red \boxtimes P_3)$ therefore $|P_4| \leq 2^{-11}$ et $P_4 = (c_5 + (red \times P_3))(1 + \varepsilon_{-210} + \varepsilon_{-230})$
- $P_5 = c_6 \boxplus (red \boxtimes P_4)$ therefore $|P_5| \leq 2^{-8}$ et $P_5 = (c_6 + (red \times P_4))(1 + \varepsilon_{-210} + \varepsilon_{-228})$
- $P_6 = c_7 \boxplus (red \boxtimes P_5)$ therefore $|P_6| \leq 2^{-5}$ et $P_6 = (c_7 + (red \times P_5))(1 + \varepsilon_{-210} + \varepsilon_{-224})$
- $P_7 = c_8 \boxplus (red \boxtimes P_6)$ therefore $|P_7| \leq 2^{-3}$ et $P_7 = (c_8 + (red \times P_6))(1 + \varepsilon_{-210} + \varepsilon_{-221})$
- $P_8 = c_9 \boxplus (red \boxtimes P_7)$ therefore $|P_8| \leq 2^{-1}$ et $P_8 = (c_9 + (red \times P_7))(1 + \varepsilon_{-210} + \varepsilon_{-219})$
- $P_9 = c_{10} \boxplus (red \boxtimes P_8)$ therefore $|P_9| \leq 1 + 2^{-10}$ et $P_9 = (c_{10} + (red \times P_8))(1 + \varepsilon_{-210} + \varepsilon_{-217})$
- $P_{10} = 1 \boxplus (red \boxtimes P_9)$ therefore $|P_{10}| \leq 1 + 2^{-9}$ et $P_{10} = (1 + (red \times P_9))(1 + \varepsilon_{-210} + \varepsilon_{-216})$
- $P_{11} = 1 \boxplus (red \boxtimes P_{10})$therefore $|P_{11}| \leq 1 + 2^{-8}$ et $P_{11} = (1 + (red \times P_{10}))(1 + \varepsilon_{-210} + \varepsilon_{-216})$

Therefore

$$res\_scs = P_{11}(1 + \varepsilon_{-209})$$

We build the polynomial such that

$$\exp(r) = (1 + r + c_{10}.r^2 + \cdots + c_0.r^{12}).(1 + \varepsilon_{-179})$$

Therefore
$|res\_scs| \leq 1 + 2^{-8}$ with
$\exp(x) = 2^k.(exp(r))^{512} = 2^k.(res\_scs.(1 + \varepsilon_{-209}).(1 + \varepsilon_{-179}))^{512}$

line 48 We perform a squaring of the result 9 times, which corresponds to raising the result to the power 512. At each iteration we perform a rounding error equal to ε_{-207}. Finally
$|res\_scs| \leq 2^3$ and
$\exp(x) = 2^k.exp(r)^{512}.(1+\varepsilon_{-170})$

line 54-60 With these lines we perform the multiplication of res_scs by 2^k. This multiplication is done by a shift on the index of $k/30$, where 30 correspond to the number of bits used within a multiprecision number. This shift is exact. Then a multiplication of res_scs by 2 to the power the rest of the euclidian division of k by 30 is done. At the end of these instructions we have:

$$\exp(x) = (res\_scs).(1+\varepsilon_{-170})$$

We are approximating the exponential with a relative error less than $2^(-170)$. This result combines with property 1, gives us the proof of correct rounding for the four rounding modes.

□

11 Analysis of the exponential

11.1 Test conditions

Table 9 lists the combinations of processor, OS and default `libm` used for our tests.

Processor	OS	compiler	default `libm`
Pentium III	Debian GNU/Linux	gcc-2.95	`glibc`, derived from `fdlibm`
UltraSPARC IIi	SunOS 5.8	gcc-2.95	Sun optimized
Xeon (Pentium 4)	Debian GNU/Linux	gcc-2.95	`glibc`, derived from `fdlibm`
PowerPC G4	MacOS 10.2	gcc-2.95	Apple specific
Itanium	Debian GNU/Linux	gcc-2.95, gcc-3.2	Intel optimized

Table 9: The systems tested

The following presents tests performed under such conditions as to suppress most of the impact of the memory hierarchy: A small loops performs 10 identical calls to the function, and the minimum timing is reported, ensuring that both code and data have been loaded in the cache, and that interruptions by the operating system do not alter the timings.

These timings are taken on random values between -745 and $+744$, which is the practical range for the exponential. We also report the timing for the worse case for the correct rounding in rounding to nearest mode of the exponential, which is $x = 7.5417527749959590085206221e - 10$.

Our libray was tuned to take into account the adequation of the evaluation scheme to the memory hierarchies of current processors (our program for the exponential evaluation uses 2.8Kbytes of table for the four rounding modes, whereas the one from *fdlibm* use 13Kbytes). However we do not have tested the impact over performance of this concern and is part of our futur works.

11.2 Results

Tables 10 gives a summary of the timings of the various libraries. The timings are normalized to the average time of the default `libm` on each system, which should be taken with care as the `libms` used by default on the tested systems are all different. Table 11 gives absolute times.

Processor	default libm # errors	time	correctly rounded libraries crlibm	MPFR	libultim
Pentium III	1/587	1	1.21	45	0.9
		1.81	39.8	93.8	3339
Itanium	1/491	1	2.6	153	1.8
		13.7	46	165.3	14499
UltraSparc IIi	1/41	1	2.66	83	1.2
		12.9	147	383	3576
PowerPC G4	1/4739	1	0.91	13.5	0.93
		1.32	9.25	26.9	1477

Table 10: Accuracy and timings for the exponential function from various libraries. Timings are normalized to the average time for the default `libm`. For each processor, the first line gives the average time, and the second line gives the worst-case time.

Processor		libm	crlibm	MPFR	libultim
Pentium III (cycles)	average time	462	562	21114	413
	overall worst-case time	837	18413	43316	66050
	correct rounding worst case time	448	15963	38969	1542415
Xeon (cycles)	average time	982	1178	24991	942
	overall worst-case time	3124	34236	138768	108920
	correct rounding worst case time	1080	30384	50436	2592660
Itanium (cycles)	average time	202	518	30995	371
	overall worst-case time	2767	9349	70660	139415
	correct rounding worst case time	131	6434	33388	2928718
UltraSparc IIi (cycles)	average time	762	2033	63570	950
	overall worst-case time	9823	112366	292129	157987
	correct rounding worst case time	292	91190	126827	2724912
PowerPC G4 (ms)	average time	2.27	2.07	30.7	2.1
	overall worst-case time	3	21	61	116
	correct rounding worst case time	2	20	59	3354

Table 11: Absolute timings for the exponential

11.3 Analysis

Processor-specific libraries

Documentation[15] from Intel labs claim to provide an exponential in only 48 cycles. This performance is possible through the wide use of non portable tricks such as inverse approximation, fused multiply and add and double extended precision. However, our tests show that the environmental cost (mainly the cost of a function call) is about 80 clock cycles! Our tests have also shown that there exists a slower path that takes up to 2767 clock cycles, which is 14 times slower. This path seems to be taken very often since the average cost is 1.5 times more expensive than the smallest execution time.

The same conclusion can be done for the mathematical library used on Ultra-SPARC IIi system, where there exists a path 13 times slower than a normal execution.

These two observations show that our two-step procedure, with a much slower second step, could be viable in the commercial world.

The mathematical library used on the PowerPC G4 with gcc is the one from Apple. This library do not provide correct rounding and is 1.1 slower than the version provided with *crlibm*. It is, however, the most accurate of the tested libraries.

The cost of correct rounding

The *libutlim* library provides correct rounding for an average cost between 0.9 (on a Pentium III) and 1.8 (on an Itanium) times the cost of the standard library. Our exponential return a result for an average cost between 0.91 (on Power-PC) and 2.66 (on Ultra-SPARC IIi) times the cost of the standard library, which is reasonable. On the other hand, MPFR provide correct rounding for an average cost between 13.5 (on Power-PC) and 153 (on Itanium) compared to the *libm*.

The main advantage of *crlibm* over *libutlim* is the upper bound on the execution time. On our tests, this bound for *crlibm* is 147 times the average *libm* cost, whereas for *libtultim* this bound goes up to 14499 times the average *libm* cost. Our two steps strategy fully benefits from knowing bounds on correct rounding worst cases.

We notice that our second step is in average 3 times faster than the multiprecision library MPFR. It shows that our multiprecision operators from *scslib*, hand tuned for 200 bits of precision perfectly fulfill the performance requirement of the second step.

Relations between the two steps of *crlibm*

For some of our test, we have disconnected the call to the second phase in the evaluation scheme and counted one miss-rounding result over 2097152 ($\approx 2^{21}$). As we can see in the first column of table 10, it mean that we are far more precise than others library.

Our second phase is 30 times slower than the first step and is called only once over 2^{13}. The cost of the second step over the average cost is:

$$\frac{1 \times (2^{13} - 1) + 30 \times 1}{2^{13}} = 1.003540039$$

which corresponds to a 0.35% overhead. This small overhead in average means that a possible performance improvment is to reduce the precision of the first step, and by the same way the number of instructions, to increase to number of time that the second step is called. It will also make the proof simpler.

12 CONCLUSION AND PERSPECTIVES

We have presented a library of elementary functions correctly rounded in double precision in the four IEEE-754 rounding modes. Although only one function is complete, we have thus shown that correct rounding can be obtained with performance (both average and worst-case) comparable to libraries without this property. Improvements over previous works include

- proven correct rounding, thanks to recent theoretical results, with detailed proofs of the code published along the code itself,
- availability of the directed rounding modes,
- bounded worst-case performance acceptable for real-time applications.

Future work include, of course, tuning and completing the library. It is obvious from our performance measurements that our first step is too accurate and too slow for a balanced average time. We will take this experience into consideration when writing first steps for other functions. The IBM library seems to get a better balance although its second and later steps are much slower. Its code, unfortunately, is little documented and difficult to prove.

Writing the proofs is a very time-consuming task, which could be partially automated for one step which is common to most function: The accumulation of error terms in order to compute the final error.

We hope that this work is a step towards making correctly rounded elementary functions a standard.

References

[1] IBM accurate portable mathematical library.

[2] LIBULTIM sun freely distributed mathematical library.

[3] MPFR, the Multiprecision Precision Floating-Point Reliable library.

[4] Open source from Intel.

[5] SCS, Software Carry-Save multiprecision mibrary.

[6] AMD. *AMD Athlon processor x86 code optimization*, august 2001.

[7] Marc Daumas and Claire Moreau-Finot. Exponential: implementation trade-offs for hundred bit precision. In *Real Numbers and Computers*, pages 61–74, Dagstuhl, Germany, 2000.

[8] D. Defour. Cache-optimised methods for the evaluation of elementary functions. Technical Report RR2002-38, LIP, École Normale Supérieure de Lyon, October 2002. Available at ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/RR/RR2002/RR2002-38.ps.gz.

[9] D. Defour. *Fonctions élémentaires: algorithmes et implémentations efficaces pour l'arrondi correct en double précision*. PhD thesis, École Normale Supérieure de Lyon,Lyon ,France, 2003.

[10] D. Defour and F. de Dinechin. Software carry-save for fast multiple-precision algorithms. In *35th International Congress of Mathematical Software.*, pages 29–40, Beijing , China, August 2002.

[11] D. Defour and F. de Dinechin. Software carry-save: A case study for instruction-level parallelism. In *7th conference on parallel computing technologies*, Nizhny-Novgorod, september 2003.

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

[13] P. M. Farmwald. High bandwidth evaluation of elementary functions. In K. S. Trivedi and D. E. Atkins, editors, *Proceedings of the 5th IEEE Symposium on Computer Arithmetic*. IEEE Computer Society Press, Los Alamitos, CA, 1981.

[14] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, Mar 1991.

[15] J. Harrison, T. Kubaska, S. Story, and P.T.P. Tang. The computation of transcendental functions on the IA-64 architecture. *Intel Technology Journal*, Q4, 1999.

[16] IEEE standard for binary floating-point arithmetic. *ANSI/IEEE Standard, Std 754-1985*, New York, 1985.

[17] R. Klatte, U. Kulisch, C. Lawo, M. Rauch, and A. Wiethoff. *C-XSC a C++ class library for extended scientific computing*. Springer Verlag, 1993.

[18] D. Knuth. *The Art of Computer Programming*, volume 2, "Seminumerical Algorithms". Addison Wesley, Reading, MA, third edition edition, 1998.

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

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

[21] R.-C. Li, P. Markstein, J. P. Okada, and J. W. Thomas. The libm library and floating-point arithmetic for HP-UX on Itanium. Technical report, Hewlett-Packard company, april 2001.

[22] P. Markstein. *IA-64 and Elementary Functions : Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000. ISBN: 0130183482.

[23] R. E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, NJ, New York, 1963.

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

[25] P. H. Sterbenz. *Floating point computation*. Prentice-Hall, Englewood Cliffs, NJ, 1974.

[26] P. T. P. Tang. Table lookup algorithms for elementary functions and their error analysis. In P. Kornerup and D. W. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 232–236, Grenoble, France, June 1991. IEEE Computer Society Press, Los Alamitos, CA.

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