

Software techniques for perfect elementary functions in floating-point interval arithmetic

Florent de Dinechin
École Normale Supérieure de Lyon
Florent.de.Dinechin@ens-lyon.fr

Sergey Maidanov
Intel Corporation
Sergey.Maidanov@intel.com

Abstract

A few recent processors allow very efficient double-precision floating point interval arithmetic for the basic operations. In comparison, the elementary functions available in current interval arithmetic packages suffer two major weaknesses. The first is accuracy, as intervals grow more in an elementary function than what is mathematically possible considering only the number format. The second is performance, well below that of scalar elementary function libraries. The difficulty is that the need to prove the containment property for interval functions usually entails sacrifices in accuracy and/or performance. This articles combines several available techniques to obtain implementations of double-precision interval elementary functions that are as accurate as mathematically possible, less than twice slower than their best available scalar counterparts, and fully verified. Implementations of exponential and natural logarithm on an Itanium[®] 2-based platform support these claims.

Keywords: Interval arithmetic, floating-point, elementary functions.

1 Introduction

The IEEE-754/IEC 60559 standard for floating-point arithmetic[3] defines floating-point formats, among which single and double precision, and specifies the behaviour of basic operators ($+$, $-$, \times , \div , $\sqrt{}$) in four rounding modes (round to the nearest or RN, towards $+\infty$ or RU, towards $-\infty$ or RD and towards 0 or RZ). The *directed* rounding modes RU, RD and RZ are intended to help implementing interval arithmetic (IA). Most processors now provide hardware support for the IEEE-754 standard, and thus the opportunity for hardware-accelerated interval arithmetic.

However, until recently, the limitations of this hardware support on most processors meant that interval arithmetic was much slower than one would think just by counting the operations. In particular, frequently changing the rounding mode had a large overhead, because processors needed to flush their FP pipeline at each mode change. This is no longer a problem on recent processors like the Intel[®] Itanium[®] processor family [5], and recent UltraSPARC[®] III processors with the VIS[™] instructions set extension by Sun Microsystems [2].

Among the next useful building blocks for efficient FP IA support is the elementary function library. This articles demonstrates the feasibility of a “perfect” elementary function interval library in IEEE-754 single and double precision. Here, perfect means

1. returning *sharp* intervals, *i.e.* intervals that are as tight as the representation mathematically allows,
2. with verified algorithms and implementations, and
3. with performance within a factor 2 of the best available corresponding scalar elementary functions.

To our knowledge, among existing libraries offering double-precision interval elementary functions (reviewed in Section 2), none offers this combination of features.

The main idea is that it takes very little to convert a *proven correctly rounded elementary function*, as developed in the `crlibm` project [1, 6], into a perfect interval function. More specifically, all the required software building blocks are already in place in the `crlibm` framework, and the proof of the containment property for the interval function can be derived straightforwardly from the proof of the correct rounding property for the function, which is available in `crlibm`. These ideas and techniques are developed in Section 2, and Section 3 discusses more specifically performance and optimisation issues, and presents some experiments on an Itanium-2 based platform. The conclusion includes a discussion on the pros and cons of sharp intervals.

2 From correctly rounded to interval functions

2.1 Floating-point elementary functions

The IEEE-754 standard requires correct rounding for $+$, $-$, \times , \div and $\sqrt{}$, but has currently no such requirement for the transcendental functions (this could change in the upcoming revision of the standard). The main reason for this is the *table maker's dilemma* (TMD): FP numbers are rational numbers, and in most cases, the image \hat{y} of a rational number x by an elementary function f is not a rational number, and can therefore not be represented exactly as an FP number. The *correctly rounded* result will be the floating-point number that is closest to this mathematical value (or immediately above or immediately below, depending on the rounding mode). A computer will evaluate an approximation y to the real number \hat{y} with accuracy $\bar{\epsilon}$, meaning that the real value \hat{y} belongs to the interval $[y/(1+\bar{\epsilon}), y/(1-\bar{\epsilon})]$. The dilemma (named in reference to the early builders of logarithm tables) occurs when this information is not enough to decide correct rounding. For instance, if $[y/(1+\bar{\epsilon}), y/(1-\bar{\epsilon})]$ contains a floating-point number z , it is impossible to decide the rounding up of \hat{y} : it could be z , or its successor.

A technique for computing the correctly rounded value, published by Ziv [24] and first implemented in IBM Accurate Portable Mathlib, is to improve the accuracy $\bar{\epsilon}$ of the approximation until the correctly rounded value can be decided. Given a function f and an argument x , a first, quick approximation y_1 to the value of $f(x)$ is evaluated, with accuracy $\bar{\epsilon}_1$. Knowing $\bar{\epsilon}_1$ and therefore $[y_1/(1+\bar{\epsilon}_1), y_1/(1-\bar{\epsilon}_1)]$, it is possible to decide if rounding y_1 is equivalent to rounding $y = f(x)$, or if more accuracy is required. In the latter case, the computation is restarted using a (slower) approximation of accuracy $\bar{\epsilon}_2$ better than $\bar{\epsilon}_1$, and so on. This approach leads to good average performance, as the slower steps are rarely taken. Besides, if the worst-case accuracy required to round correctly a function is known [14], then only two steps are needed. This improvement, implemented in the `crlibm` project [1, 6], makes it easier to optimise and prove each step. Here the proof is mostly a computation, for each step, of a sharp bound $\bar{\epsilon}$ on the total error of the implementation (due to accumulated approximation and rounding errors). In the first step, this bound is used to decide whether to go for the second step. In the second step, it is enough that this bound is lower than the worst-case accuracy to guarantee correct rounding. Note that this technique works for obtaining correctly rounded values for any of the IEEE-754 rounding modes.

2.2 FP interval elementary functions

The containment property for elementary functions typically requires the computation of the images of both ends of the input interval, one with rounding up, the other with rounding down, depending on the monotonicity of the function. For functions which are not monotonic on their input interval, the situation may be more complex. However, elementary functions, by definition, have useful mathematical properties which can be exploited. For instance, scalar implementations of periodic functions always begin with a range reduction which brings the argument in a selected period. From the information computed in this range reduction, one may deduce the monotonicity information needed to manage interval functions [18].

2.3 A matter of proof

For intervals to guarantee *validated* numerics, it is crucial to provide an extensive proof of the containment property, which again boils down to the proof of an error bound. The best examples so far are the proof [12] of the elementary functions of C-XSC/`fi_lib/filib++`¹, and that of `crlibm` [1]. The first implement error computation using their own interval arithmetic package, the second use Maple and, more recently, the Gappa proof assistant [7]. The main difference is that the errors computed for `crlibm` are much tighter (in the order of 2^{-60} , versus 2^{-51} to 2^{-53} for the `fi_lib` family).

Both proofs rely on external tools to compute the approximation error of a Remez polynomial, which may be considered a weakness in the proof as these tools are not proven themselves. Harrison [11] computes a machine-verified bound of such approximation error in two step, first approximating the exponential with a suitable Taylor series, then proving a bound on the distance between the Remez and the Taylor polynomials – a tractable but non trivial task. An easier (but less efficient) approach is used by Ershov and Kashevarova [9] for the mathematical library of LEDAS Math solver². They directly evaluate the function using Taylor or Chebychev series whose coefficients have been rounded in the proper direction. A detailed proof of the implementation is missing, but [9] is a convincing sketch.

Another approach was used by Rump for INTLAB [20] to capitalise on existing `libm` implementations, which are highly optimised and accurate. Rump’s algorithm computes the value of the function as a small deviation (given by a small polynomial) with respect to some reference point (computed by zeroing the least significant bits of the mantissa of the input number). The maximum error due to this polynomial is easy to bound. The algorithm then adds the deviation to the value of the function at the reference point which is given by a call to the standard `libm`. The maximum error of the `libm` function with respect to the true value has been precomputed beforehand (only once) over all reference points, which allows to bound the total error of this evaluation scheme. With respect to an implementation of the function from scratch, this scheme is a compromise between performance and ease of proof: The code will be slower and the result less accurate, because it adds computations to those performed by the `libm`. However, the proof will be much easier (the evaluation error of a small polynomial). However, comparing Rump’s proofs [20] and the *from scratch* approaches [9, 12, 7] seems to indicate that this approach saves less than half the proof work.

To our knowledge, there is no published proof of the implementations of elementary functions of other leading FP IA packages. We assume that the interval functions in Sun development environments derive from those published by Priest [18]. PROFIL/BIAS³ uses calls to the standard mathematical library with safety margins added. This does not provide a validated solution: Standard libraries are not proven and current standards do not even specify their accuracy.

2.4 Accuracy versus performance

From the *accuracy* point of view, the libraries by Priest [18] and Ershov and Kashevarova [9] will return a sharp interval with good probability, otherwise the interval will lose at most one ulp on each bound. In contrast, INTLAB, the `fi_lib` family and PROFIL/BIAS will almost always return an interval where each bound is several ulps away from the ideal one. The reason for this loss of accuracy is the use of the target precision for the intermediate computations, which entails that several ulps are lost in the process (to rounding or approximation errors). Priest [18] uses classical table-based techniques [22, 23, 15] to reach an accuracy better than 1.5 ulps. There is a trade-off between accuracy and performance here: Priest mentions earlier works [4, 13] which come closer to 1-ulp accuracy at the expense of performance, as a larger working precision has to be emulated.

¹<http://www-info2.informatik.uni-wuerzburg.de/staff/wvg/Public/filib++.html>

²www.ledas.com

³<http://www.ti3.tu-harburg.de/Software/PROFILEnglisch.html>

From the *performance* point of view, all the libraries mentioned have performance within a factor 20 of the `libm`, their scalar equivalent. The most efficient library by far is that of Priest, who remarked [18] that a monolithic interval function should be faster than two successive calls to the scalar function: The computation of the two endpoints is intrinsically parallel, allowing for more efficient use of modern super-scalar, deeply pipelined processors than two successive calls to one function. Besides, computing both endpoints in parallel allows to share some of the burden of memory accesses for table look-ups.

2.5 Techniques for the perfect interval library

To sum up the previous discussion, a perfect interval library may be obtained by

1. parallel evaluation of both endpoints to ensure efficient pipeline usage, as suggested by Priest,
2. a two-step approach *à la* Ziv for each endpoint to ensure sharpness, and nevertheless performance,
3. and a comprehensive proof as first shown in `fi.lib`, but using the automated and performance-driven techniques of the `crlibm` framework.

A difficulty is that, as far as performance is concerned, points 1 and 2 conflict: The two-step approach means tests and branches, which have to be laid out carefully in order to allow a streamlined parallel execution. This question is studied in Section 3.

2.6 When worst cases are missing

It is actually easier to write a perfect interval function than a correctly rounded scalar one. Indeed, the worst-case accuracy required for correct rounding is still missing for many functions on many intervals⁴. For some intervals and some functions (most notably the trigonometric functions for large arguments), it is known that current techniques for computing worst case accuracy will not work [14]. This prevents us from writing a two-step proven correctly rounded function, however we may still design an interval function which will be verified, efficient, and tight with a very high probability. The idea is to test, at the end of the second step, if any of the interval bounds being computed is difficult to round, and to enlarge the returned interval by one ulp only in this case.

Will such a case happen? According to statistical arguments by Gal [10, 17], the worst case accuracy required for correct rounding the trigonometric functions, on the domain where worst cases are missing, is expected to be about 2^{-117} . Current `crlibm` code for these functions is accurate to 2^{-124} . Therefore, the probability that there exist a floating-point input argument for which the previous test would succeed (and require to enlarge the interval) is roughly of 2^{-7} . In other words, a `crlibm`-based implementation will be *probably perfect*. In any case, the numerical effect of this possible enlargement can be disregarded, and the containment property will provably not be violated.

Another option would be to launch a higher precision computations, should such a case happen, but it is difficult to justify writing, proving and maintaining code that is very probably useless.

Note that the example functions presented in the sequel do not have this problem, since their worst case accuracy is known. Note also that in the example of the trigonometric of a large argument, a quick test following argument reduction may determine if the input interval contains a period of the function, in which case the output interval is simply $[-1, 1]$. Here the cases where we are currently unable to compute the worst-case accuracy are also the cases where the interval function is most likely to be degenerate, and this is not a coincidence.

From a performance point of view, this additional test may be fairly expensive: One needs to consider all the bits between the 53rd (round bit) and the last significant bit (the 124th in current implementation), and check whether they are all zeroes or all ones. However, this test will only happen in the second step, therefore its average performance impact will be negligible [6].

⁴<http://perso.ens-lyon.fr/jean-michel.muller/Intro-to-TMD.htm>

3 Experiments and results

In this section, an Itanium-2 based platform was chosen because it provides the best hardware support for interval arithmetic among currently available processors. The exponential and logarithm functions were chosen because their worst-case accuracy for correct rounding in double precision is known, and because scalar implementations optimised for the Itanium-2 processor of such correctly rounded functions were already available [6, 8].

3.1 An exponential and a logarithm function

Space prevents us to detail the algorithms used to evaluate the functions. They are exactly the same as their scalar counterparts in the `crlibm` distribution using double-extended arithmetic [6, 1], which itself was inspired by previous work [21, 16, 15]. We concentrate here on specific work for turning these scalar functions into interval ones.

Both functions are implemented in two Ziv steps: the first step handles exceptional cases (zeros, sub-normals, infinities) and evaluates in parallel the value of the function at both endpoints, using double-extended precision (64 bits of mantissa) with round to nearest mode. As some accuracy is lost to the rounding error, these first steps are accurate to $\bar{\epsilon} = 2^{-61}$ for both functions (a comprehensive proof of this error bound is also available in the `crlibm` CVS repository). This means that correct rounding can be deduced from this intermediate result with a probability of about $1 - 2^{-61+53+1}$, or in more than 99% of the cases.

3.2 Directed rounding on the Itanium

The Itanium processors can mix different rounding modes and precisions at no cost thanks to the availability of 4 *floating-point status registers* (FPSR), selected on a per-instruction basis [5]. Out of these, SF0 is usually preset to RN in double, and SF1 to RN in double-extended. We chose to use SF3, preset as “round up to double precision”, to implement directed rounding. Round down is obtained by rounding up the opposite, in order to use only one of the extra FPSR (another possible use for these registers is speculation). Figure 1 gives some macros, using assembler intrinsics accepted by the Intel and HP C compilers, that implement round up and down of a double-extended number (for the first step) and a double-double-extended number (or DDE, for the second step) [6]. These macros use *fused multiply and add* (FMA) operations, which compute $a * b + c$ with only one rounding.

3.3 Parallel evaluation

Writing two parallel paths is straightforward. Figure 2 gives the code of the polynomial evaluation of the logarithm according to Estrin’s scheme [17]. Suffixes `i` and `s` denote variables corresponding to the the lower and upper bounds of the interval. All the variables are double-extended.

The Itanium-2 processor provides two parallel FMAs with 4-cycle latency. Estrin’s scheme is particularly suited to such a processor [21, 5] because it exposes parallelism which improves pipeline usage. Running two evaluations in parallel improves pipeline efficiency further.

```
#define ROUND_EXT_TO_DOUBLE_UP(x)  _Asm_fma(_FR_D, 1.0, x, 0.0, _SF3)    // 1*x + 0
#define ROUND_EXT_TO_DOUBLE_DOWN(x) -_Asm_fma(_FR_D, -1.0, x, 0.0, _SF3)
#define ROUND_DDE_TO_DOUBLE_UP(xh,xl)  _Asm_fma(_FR_D, 1.0, xh, xl, _SF3) // 1*xh + xl
#define ROUND_DDE_TO_DOUBLE_DOWN(xh,xl) -_Asm_fma(_FR_D, -1.0, xh, -xl, _SF3)
```

Figure 1: Rounding macros for Itanium

<code>z2i = zi*zi;</code>	<code>z2s = zs*zs;</code>
<code>p67i = c6 + zi*c7;</code>	<code>p67s = c6 + zs*c7;</code>
<code>p45i = c4 + zi*c5;</code>	<code>p45s = c4 + zs*c5;</code>
<code>p23i = c2 + zi*c3;</code>	<code>p23s = c2 + zs*c3;</code>
<code>p01i = logirhi + zi*c1;</code>	<code>p01s = logirhs + zs*c1;</code>
<code>z4i = z2i*z2i;</code>	<code>z4s = z2s*z2s;</code>
<code>p47i = p45i + z2i*p67i;</code>	<code>p47s = p45s + z2s*p67s;</code>
<code>p03i = p01i + z2i*p23i;</code>	<code>p03s = p01s + z2s*p23s;</code>
<code>p07i = p03i + z4i*p47i;</code>	<code>p07s = p03s + z4s*p47s;</code>
<code>logi = p07i + Ei*log2h;</code>	<code>logs = p07s + Es*log2h;</code>

Figure 2: Parallel polynomial evaluation

Obviously, such a scheme will require twice as many registers as a sequential one. This is not a problem on Itanium, however. Ideally, this should not be our concern: a programmer should expose as much parallelism as possible, and leave it to the compiler to exploit it or not, depending on the capabilities of the target processor.

A reviewer remarked that there is a lot of redundant work in these parallel evaluations as soon as the input interval is tight enough. Unfortunately, we see no way of exploiting that to improve code efficiency: It would require to add tests to the code, which will slow down the average case. We considered testing the special case of a point interval, as is done in `fi_lib`, and even there felt that the benefit (less than 50% improvement for point intervals) was not worth the overhead to the general case (about ten cycles).

3.4 Combined rounding test

As multiple dependent tests may be expensive in a deep pipeline, both tests for correct rounding are merged into one single test. These tests consist in extracting the bits of the mantissa between the 53rd (round bit for double-precision) and the 61st (last significant bit considering the accuracy of the computation), and checking whether they are either all zeroes or all ones, which would mean that the number is very close to a double-precision number [14]. Other possibilities exist for this test, but this approach, using only 64-bit integer arithmetic which is very fast on the Itanium, was found the most effective.

Figure 3 gives the corresponding code for the logarithm. Most variables are 64-bit integers, except `logi` and `logs` which are double-extended, and `result` which is an interval of doubles. The `GET_EXT_MANTISSA` macro extracts the 64-bit mantissa of a double-extended number (its implementation is processor-dependent). The `log_accurate` function performs the second step in double-double-extended arithmetic, and returns a pair of double-extended numbers which are then added together to a double with rounding up.

Exponential is similar, with the exception of a trick that makes it slightly less readable: The last operation of the chosen algorithm is a multiplication by a power of two. This operation being exact in IEEE-754 arithmetic, the rounding test can be performed on the intermediate result before this last multiplication, so that it runs in parallel with it.

3.5 A note about portability

The code of Figures 2 and 3, as well as the code of the `log_accurate` function, no longer contains any Itanium-specific code: to compile it on an x86-compatible processor, all it takes is redefine implementations of the macros. The proof depends on lemmas that describe the behaviour of the macros, so these lemma also have to be re-proven, which is very simple here. The rest is 64-bit integer arithmetic which is supported by compilers on 32-bit machines, although possibly in a

```

/* Tentatively set the result */
result.INF = ROUND_EXT_TO_DOUBLE_DOWN(logi);
result.SUP = ROUND_EXT_TO_DOUBLE_UP(logs);

/* Now check correct rounding using 64-bit arithmetic on the mantissas */

mantissai = GET_EXT_MANTISSA(logi);
mantissas = GET_EXT_MANTISSA(logs);
bitsi = mantissai & (0x7ff&(accuracymask));
bitss = mantissas & (0x7ff&(accuracymask));
infDone= (bitsi!=0) && (bitsi!=(0x7ff&(accuracymask)));
supDone= (bitss!=0) && (bitss!=(0x7ff&(accuracymask)));

/* Only one test, expected true */
if(__builtin_expect(infDone && supDone, TRUE))
    return result;

/* otherwise launch accurate computation */
if(!infDone) {
    log_accurate(&th, &t1, zi, Ei, indexi);
    result.INF = ROUND_DDE_TO_DOUBLE_DOWN(th,t1);
}
if(!supDone) {
    log_accurate(&th, &t1, zs, Es, indexs);
    result.SUP = ROUND_DDE_TO_DOUBLE_UP(th,t1);
}

```

Figure 3: Combined rounding test

sub-optimal way. Previous experience with the scalar logarithm of `crlibm`⁵ suggests that with such a macro-based approach, portability, provability and efficiency do not necessarily conflict.

3.6 Accuracy and performance

Absolute performance results are given in Table 1. One call to our interval function is faster than two calls to the corresponding point functions with directed rounding, as predicted by Priest [18]. The proposed implementation is comparable to twice the default scalar `libm` (which does not pay the price of a function call since it is inlined by the compiler).

Table 2 evaluates the performance and accuracy of the following iteration:

$$\begin{cases} y_{n+1} = \log(x_n) \\ x_{n+1} = e^{y_{n+1}} \end{cases} \quad (1)$$

It is easy to check that the initial interval size should grow by exactly two ulps at each iteration, in the case of a sharp implementation. This expected behaviour is observed on our implementation. The growth of 13 ulps per iteration of the `fi_lib`-based implementations is consistent with [12].

Library	exp	interval exp	log	interval log
<code>libm</code>	42	n/a	31	n/a
<code>fi_lib</code>	586	1038	619	1158
<code>crlibm</code>	60	69	66	96

Table 1: Compared performance results on Itanium 2 processor (timings in cycles).

⁵See the `log-de.c` file in the `crlibm` distribution since version 0.10beta. The macros are defined in `double-extended.h`

Library	Cycles	Ulps
<code>fi_lib</code>	2215	13
<code>crlibm</code> , RU and RD functions	342	2
<code>crlibm</code> merged interval function	249	2

Table 2: Average performance in cycles and interval growth in ulps of iteration (1) on Itanium 2

Experimenting around this code, we found that the data structure used for intervals in `fi_lib` has a significant overhead, which explains that the timing of a loop (which also performs some bookkeeping) is much higher than the sum of the timings of the functions in Table 1.

4 Concluding remarks

This article surveyed and demonstrated software techniques available for the implementation of “perfect” interval floating-point elementary functions, where perfect means tight, efficient and fully verified. It also remarked that it is possible to write a probably-perfect interval function even in the cases where theoretical results are missing to write a proven correctly rounded function. In such cases, the function will still be efficient and verified, but might very rarely (and probably never) return an interval that is not the tightest possible.

As the presented implementations are sharp, there can be no further improvement in accuracy. Performance, however, can still be improved. The main issue is the organisation of the tests for special cases and correct rounding. Present code is slow when handling intervals where a bound is infinity or zero, for instance. Whether this is a problem will depend on the application. In principle, infinities occur often in the development of an IA application, but should occur rarely in production.

The functions presented in this paper are still experimental and non-portable, and a lot of work will be needed to develop a full portable library (needed to compare to Sun’s current offering, for example). The `crlibm` framework is striving to encapsulate processor-specific optimisations into common C macros selected at compile time. A conclusion of the present study is that interval functions can build upon this framework and extend it.

Revol and Rouiller [19] and Kahan⁶ have advocated the use of multiple-precision interval arithmetic. The idea is to perform the bulk of computations in native precision, and increase the precision only when needed due to interval bloat. A perfect interval library as described in this article is a building block for implementing such efficient strategies.

In defence of sharp bounds

As a conclusion, one might discuss the usefulness of providing sharp bounds in interval elementary functions.

On one side, numerical portability of an application (the fact that it will return the same answer whatever the system) is probably not so much of an issue as soon as interval arithmetic is used: the result will be validated nevertheless. Our sharp library will from time to time return an interval smaller by one or two ulps than Priest’s or Ershov’s. From a practical point of view, this accuracy improvement is probably of little significance. An interval library implemented with the accuracy standards of current scalar `libms` [21, 15] would typically bloat the intervals by one ulp in one percent of cases only, and have even better performance than the present approach.

On the other side, the points in favour of sharp bounds are the following:

- It is more elegant to always return the best possible interval considering the number representation.
- It is consistent with the existing IEEE-754 standard.

⁶www.cs.berkeley.edu/~wkahan/Mindless.pdf

- It allows for fair benchmarking where one compares functionally equivalent programs.
- It entails small performance penalty in average.

Does it entail more development work? Not really, since even a non-sharp interval function needs a proof of its error bounds. Experiments conducted in `crlibm` consistently show [6] that the proof of the second step is much easier because it doesn't include the argument reduction (already proven in the first step), and it needn't be optimised as tightly as the first step.

Thus, the cost of sharp bounds is an increase in code size, but negligible performance overhead and much less than doubling the proof effort. The benefits, small as they may be, may be worth the effort.

References

- [1] CR-Libm, a library of correctly rounded elementary functions in double-precision. <http://lipforge.ens-lyon.fr/www/crlibm/>.
- [2] Interval arithmetic in high performance technical computing. Technical report, Sun Microsystems, September 2002.
- [3] ANSI/IEEE. *Standard 754-1985 for Binary Floating-Point Arithmetic (also IEC 60559)*. 1985.
- [4] K. Braune. Standard functions for real and complex point and interval arguments with dynamic accuracy. In *Scientific Computation with automatic result verification*, pages 159–184. Springer-Verlag, 1988.
- [5] M. Cornea, J. Harrison, and P.T.P Tang. *Scientific Computing on Itanium-based Systems*. Intel Press, 2002.
- [6] F. de Dinechin, A. Ershov, and N. Gast. Towards the post-ultimate libm. In *17th Symposium on Computer Arithmetic*, pages 288–295. IEEE Computer Society Press, June 2005.
- [7] F. de Dinechin, Ch. Q. Lauter, and G. Melquiond. Assisted verification of elementary functions. Technical Report RR2005-43, LIP, September 2005.
- [8] F. de Dinechin, Ch. Q. Lauter, and J.-M. Muller. Fast and correctly rounded logarithms in double-precision. *Theoretical Informatics and Applications*, 2006. To appear.
- [9] A. G. Ershov and T. P. Kashevarova. Interval mathematical library based on Chebyshev and Taylor series expansion. *Reliable Computing*, 11(5):359–367, 2005.
- [10] S. Gal. Computing elementary functions: A new approach for achieving high accuracy and good performance. In *Accurate Scientific Computations, LNCS 235*, pages 1–16. Springer Verlag, 1986.
- [11] J. Harrison. Floating point verification in HOL light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory, 1997.
- [12] W. Hofschuster and W. Krämer. FLIB, eine schnelle und portable Funktionsbibliothek für reelle Argumente und reelle Intervalle im IEEE-double-Format. Technical Report Nr. 98/7, Institut für Wissenschaftliches Rechnen und Mathematische Modellbildung, Universität Karlsruhe, 1998.
- [13] W. Krämer. Inverse standard functions for real and complex point and interval arguments with dynamic accuracy. In *Scientific Computation with automatic result verification*, pages 185–211. Springer-Verlag, 1988.

- [14] V. Lefèvre, J.-M. Muller, and A. Tisserand. Towards correctly rounded transcendentals. *IEEE Transactions on Computers*, 47(11):1235–1243, November 1998.
- [15] 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.
- [16] P. Markstein. *IA-64 and Elementary Functions: Speed and Precision*. Hewlett-Packard Professional Books. Prentice Hall, 2000. ISBN: 0130183482.
- [17] J.-M. Muller. *Elementary Functions, Algorithms and Implementation*. Birkhauser, Boston, 1997/2005.
- [18] D.M. Priest. Fast table-driven algorithms for interval elementary functions. In *13th IEEE Symposium on Computer Arithmetic*, pages 168–174. IEEE, 1997.
- [19] N. Revol and F. Rouillier. Motivations for an arbitrary precision interval arithmetic and the MPFI library. In *Workshop on Validated Computing*, pages 155–161. SIAM, 2002.
- [20] S. M. Rump. Rigorous and portable standard functions. *BIT Numerical Mathematics*, 41(3), 2001.
- [21] S. Story and P.T.P. Tang. New algorithms for improved transcendental functions on IA-64. In *14th IEEE Symposium on Computer Arithmetic*, pages 4–11. IEEE, April 1999.
- [22] P.T.P. Tang. Table-driven implementation of the exponential function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 15(2):144–157, June 1989.
- [23] P.T.P. Tang. Table-driven implementation of the logarithm function in IEEE floating-point arithmetic. *ACM Transactions on Mathematical Software*, 16(4):378 – 400, December 1990.
- [24] A. Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, September 1991.