Assisted verification of elementary functions using Gappa

Florent de Dinechin
LIP, projet Arénaire
ÉNS-Lyon
46 allée d'Italie,
69364 Lyon Cedex 07, France
Florent.de.Dinechin@ens-lyon.fr

Christoph Quirin Lauter
LIP, projet Arénaire
ÉNS-Lyon
46 allée d'Italie,
69364 Lyon Cedex 07, France
Christoph.Lauter@ens-lyon.fr

Guillaume Melquiond
LIP, projet Arénaire
ÉNS-Lyon
46 allée d'Italie,
69364 Lyon Cedex 07, France
Guillaume.Melquiond@ens-lyon.fr

ABSTRACT

The implementation of a correctly rounded or interval elementary function needs to be proven carefully in the very last details. The proof requires a tight bound on the overall error of the implementation with respect to the mathematical function. Such work is function specific, concerns tens of lines of code for each function, and will usually be broken by the smallest change to the code (e.g. for maintenance or optimization purpose). Therefore, it is very tedious and error-prone if done by hand. This article discusses the use of the Gappa proof assistant in this context. Gappa has two main advantages over previous approaches: Its input format is very close to the actual C code to validate, and it automates error evaluation and propagation using interval arithmetic. Besides, it can be used to incrementally prove complex mathematical properties pertaining to the C code. Yet it does not require any specific knowledge about automatic theorem proving, and thus is accessible to a wider community. Moreover, Gappa may generate a formal proof of the results that can be checked independently by a lower-level proof assistant like Coq, hence providing an even higher confidence in the certification of the numerical code.

1. INTRODUCTION

Computing floating-point elementary functions with correct rounding [12, 2] requires to be able to prove a bound on the overall evaluation error. Moreover, this bound should be tight, as a loose bound will have a negative impact on performance [4]. Similarly, proving the containment property for an interval elementary functions requires computing an error bound on the evaluation [7]. This bound should also be tight, as a looser bound means returning a larger interval result, and hence useless interval bloat.

This article describes an approach to machine-checkable proofs of such tight error bounds that is both interactive and easy to manage, yet much safer than a hand-written proof. The novelty here is the use of a tool that transforms a high-level description of the proof into a machine-checkable version, in contrast to previous work by Harrison [6] who directly described the proof of the implementation of an exponential function in all the low-level details. The Gappa approach is more concise and more flexible in the case of a subsequent change to the code. More importantly, it is accessible to people outside the formal proof community.

An extended version of this article is available as LIP research report 2005-43 [3]. Next section describes the challenges posed by automatic computation of tight error bounds. Section 3 describes the Gappa tool. Sections 4 and 5 give an overview on the techniques for proving an elementary function using Gappa and give an extensive example of the interactive construction of the proof.

SAC'06 April 23-27, 2006, Dijon, France

2. COMPUTING A TIGHT ERROR BOUND

The evaluation of an elementary function is classically performed by a polynomial approximation valid on a small interval only. A *range reduction* step brings the input number x into this small interval, and a *reconstruction* step builds the final result out of the results of both previous steps. For example, the logarithm may use as a range reduction the errorless decomposition of x into its mantissa m and exponent E: $x = m \cdot 2^E$. It may then evaluate the logarithm of the mantissa as a polynomial, and the reconstruction consists in evaluating $\log(x) \approx \log(m) + E \cdot \log(2)$.

When an intermediate precision larger than the native one is needed, implementations use *double-extended* arithmetic on processors that support it, or *double-double* arithmetic, where a number is held as the unevaluated sum of two doubles [5, 8].

The evaluation of an elementary function using such algorithms entails two main sources of errors.

- Approximation errors (also called methodical errors), such as the error of approximating a function with a polynomial. One may have a mathematical bound for them (given by a Taylor formula for instance), or one may have to compute such a bound using numerics (for minimax polynomials for instance).

- Rounding errors, produced by most floating-point operations of the code.

Many floating-point operations are exact, and the experienced author of floating-point code will try to use them. Examples include multiplication by a power of two, subtraction of numbers of similar magnitude thanks to Sterbenz' Lemma, exact addition and exact multiplication algorithms

(returning a double-double), multiplication of a small integer by a floating-point number whose mantissa ends with enough zeroes (used in Cody-Waite range reduction [11]), etc.

However, an optimized elementary function implementation will stack approximation over approximation to avoid computing more accurately than strictly needed. It then takes considerable discipline to define properly what is the error of what with respect to what.

Thus, the difficulty of evaluating a tight bound on an elementary function implementation is to combine all these errors without forgetting any of them, and without using overly pessimistic bounds when combining several sources of errors. The typical trade-off here will be that a tight bound requires considerable more work than a loose bound (and its proof might inspire considerably less confidence).

As an illustration, proofs written for version of the `crlibm` project[1] up to version 0.8 are typically composed of several pages of paper proof and several pages of supporting Maple for a few lines of code. This provides an excellent documentation and helps maintaining the code, but experience has consistently shown that such proofs are extremely error-prone. Implementing the error computation in Maple was a first step towards the automation of this process, but if it helps avoiding computation mistakes, it does not prevent methodological mistakes. Gappa was designed in order to fill this void.

3. THE GAPPA TOOL

Gappa[2] is a tool that extends the interval arithmetic paradigm to the field of numerical code certification [1]. Basically, Gappa's purpose can be summarized with the example of this logical property: "$x+1 \in [2,3] \Rightarrow x \in [?,?]$". The interrogation marks mean that the interval is not defined and it will be up to the tool to find a range for the expression x such that the property holds. Gappa works on mathematical expressions on real numbers and will find $[1,2]$. By using interval arithmetic to evaluate this range, Gappa can easily transform its computations in a formal proof of the whole property. This proof is completely independent from Gappa and its validity does not depend on Gappa's own validity.

Gappa was initially designed to compute the range of floating-point variables in a program and the rounding error they suffer from. Unfortunately this approach was excessively limiting and prevented certifying more complex numerical codes. Gappa was then modified so that it could try to bound any mathematical expressions on real numbers and formally prove them [10]. Since the original purpose was to certify floating-point applications, the system of rounding operators was introduced so that these mathematical expressions can precisely express the floating-point values of variables.

Gappa uses a base of theorems on real arithmetic and of theorems on rounding operators, e.g. absolute and relative errors of floating-point computations. It also uses a base of theorems in order to rewrite mathematical expressions so as to obtain tight intervals for logical properties that commonly appear when certifying numerical code. This way of computing intervals was then fine for writing robust floating-point geometric predicates, but still not good enough for certifying the optimized code of a correctly-rounded floating-point elementary function.

Consequently, Gappa was modified so that the user could provide additional rewriting rules to Gappa's engine. This was necessary because the tool is unable to guess the optimizations the developer did: truncated series, neglected terms, multi-precision arithmetic, and so on. Section 5 will show on the example of the logarithm that this enhancement of Gappa's engine was enough for it to handle the complexity of elementary functions.

[1] `http://lipforge.ens-lyon.fr/www/crlibm/`
[2] `http://lipforge.ens-lyon.fr/www/gappa/`

4. PROVING ELEMENTARY FUNCTIONS USING GAPPA

As in every proof work, style is important when working with Gappa: in a machine-checked proof, bad style will not in principle endanger the validity of the proof, but it may prevent its author to get to the end. In the `crlibm` framework, it may hinder acceptance of machine-checked proofs among new developers.

This section is an attempt to describe the approach used in `crlibm`. It may be inadequate for applications other than elementary functions, and even for elementary functions it might be improved further. It consists in three steps, which correspond to the three sections of a Gappa input file.

- First, the C code is translated into Gappa equations, in a systematic way that ensures that the Gappa proof will indeed prove some property of this program (and not of some other similar program). Then equations are added describing what the program is supposed to implement. Usually, these equations are also in correspondence with the code.

- Then, the property to prove is added. It is usually in the form `hypotheses -> properties`, where the hypotheses are known bounds on the inputs, or contribution to the error determined outside Gappa, like the approximation errors.

- Finally, one has to add *hints* which indicate to the Gappa engine how to unroll the proof, or make explicit the implicit knowledge one has about the code. This last part is built incrementally.

The following details these three steps.

4.1 Notation conventions

Before starting, one has to remember that for Gappa there is only one type of variables, which may hold arbitrary intervals. In the proof of an elementary function, we use the following conventions: Gappa variables behaving exactly like the C variables have exactly the same name, which should begin with a lower case letter. Variables for mathematically ideal terms begin with a capital "`M`". All the other intermediate variables will begin with capital letters. In addition, related variables should have related and, wherever possible, explicit names.

These conventions are best explained with an example: Consider the following code bit, extracted from the proof of Section 5.

```
Mul12(&zhSquareh, &zhSquarel, zh, zh);
zhCube = zh * zhSquareh;
```

It inputs a variable `zh`, computes exactly its square as a double-double `zhSquareh` + `zhSquarel` using Dekker's algorithm (here implemented as a call to the `Mul12` function), then computes an approximation to its cube. To analyse this code, we will need at least one variable `MZCube`, which is the mathematical value that `zhCube` intends to approximate in this code, and one variable `MZSquare`.

Now the notion of "mathematically ideal" may be quite subtle. In our example, `zh` is itself an approximation to an ideally reduced argument, noted of course `MZ`. Therefore, it should be clear that the equation defining `MZSquare` is `MZSquare = MZ * MZ` and not `MZSquare = zh * zh`: Although the squaring was exact in the code, it did not compute the square of the exact reduced value.

Again, these are conventions and are part of our proof style, not part of Gappa syntax: the capitalization will give no information to the tool, and neither will the fact that variables have related names.

Another useful convention will be to define variables for absolute and relative errors beginning respectively with `delta` and `epsilon`, as in the following example:

```
deltaZh = zh - MZ;
epsilonZhCube = (zhCube - MZCube) / MZCube;
```

This last convention makes the proofs much more readable and eases the task of writing hints, especially when dealing with relative errors.

4.2 Translating a FP program

If the C code is itself sufficiently simple and clean, this step only consists in making explicit the rounding operations that are implicit in the C source code. The syntax `float<ieee_64,ne>(`*Expr*`)` corresponds to a rounding to the nearest double of *Expr*. For instance, if `a` and `b` are floating-point variables, `float<ieee_64,ne>(a+b)` is the IEEE-754-compliant addition with correct rounding. For the sake of clarity, this rounding operator will now be written as the shorter identifier `float64ne`.

Adding by hand all the rounding operators, however, would be tedious and error-prone, and would make the Gappa syntax so different from the C syntax that it would degrade confidence and maintainability. Besides, one would have to apply without error the rules (well specified by the C99 standard) governing implicit parentheses in a C expression. For these reasons, Gappa has a syntax that instructs it to perform this task automatically, illustrated by the following example: The C line

```
q = c3 + zh * (c4 + zh * (c5 + zh * (c6 + zh *
    c7)));
```

and the Gappa line

```
q float64ne= c3 + zh * (c4 + zh * (c5 + zh * (
    c6 + zh * c7)));
```

define the same mathematical relation between their right-hand side and left-hand side, under the conditions that all the C variables are double-precision variables, that the Gappa variables on the right-hand side imitate them (see the lowercase convention), and also, of course, that the compiler/OS/processor combination used to process the C code respects the C99 and IEEE-754 standards and computes in double-precision arithmetic.

All this means that for straight-line program segments with mostly double-precision variables, a set of corresponding Gappa definitions can be obtained straightforwardly by just replacing the C `=` with Gappa `float64ne=`, a very safe operation.

There are other syntaxes [3]. For example one may express properties of double-double operators, for which a bound on the relative error is known since Dekker [5, 9], but proven outside Gappa.

4.3 Defining ideal values

The next operation to carry out is to define in Gappa what the C code is supposed to implement. For instance, using our previous conventions, the line for `q` was probably evaluating the value of the same polynomial of the ideal `MZ`:

```
MQ = c3 + MZ * (c4 + MZ * (c5 + MZ * (c6 + MZ *
     c7)));
```

We have kept the polynomial coefficients in lower case: The polynomial thus defined nevertheless belongs to the set of polynomial with real coefficients, and we know how to compute in Maple a bound of its relative error with respect to the function it approximates.

Another question is, how do we define the mathematical function? Gappa has no builtin sine or logarithm. The current approach can be described in English as: "$\log(1+Z)$ is a value which, if Z is smaller 2^{-8}, is within a relative distance of 2^{-63} of our ideal polynomial". In Gappa, this translates to hypotheses in the property to prove (with the Gappa syntax `1b-8` for 2^{-8}):

```
Z in [-1b-8,1b-8] /\
(MP - MLog1pZ) / MLog1pZ  in [-1b-63, 1b-63]
```

Here the interval of Z is defined by the range reduction, and the 2^{-63} bound has to be computed outside Gappa (for instance thanks to an infinite norm evaluated in Maple). This is in principle a weakness of the proof, however we take some safety margins, and on the considered intervals, elementary functions are regular enough to trust Maple's infinite norm.

Then we may use the reconstruction associated with the argument reduction used to define the mathematical function on the whole interval, as for the logarithm:

```
MLogx = MLog1pZ + E * MLog2;
```

4.4 Defining the property to prove

The theorem to prove is expressed as implications using classical first-order logic, with some restrictions. In practice we usually list a conjunction of hypotheses, the `->` operator, and a conjunction of conclusions to prove. For a full example, see next section.

4.5 Hints

The hint part reflects the work humans still must do in order to prove the numerical properties of the code.

The hints have the following form:

```
Expr1 -> Expr2;
```

which is used to give the following information to Gappa: "I believe for some reason that, should you need to compute an interval for $Expr_1$, you might get a tighter interval by trying the mathematically equivalent $Expr_2$". This fuzzy formulation is better explained by considering the following examples.

1. The "some reason" in question will typically be that the programmer knows that variables x_h, MX and X are different approximations of the same quantity, and furthermore that x_h is an approximation to X which is an approximation to MX. Suppose that at some point Gappa has to compute $x_h - MX$, and even that it already has a good interval for x_h and a good interval for MX (the values will be quite similar since x_h approximates MX). In this case, standard interval arithmetic will lead to a very coarse interval for $x_h - MX$.

The adequate hint to give in this case is

```
xh - MX -> (xh - X) + (X - MX);
```

It will instruct Gappa to first compute intervals for $x_h - X$ and $X - MX$ (both of which will be small) and sum them to get an interval for $x_h - MX$ (which will thus be tight as well).

Note that if one defines `delta*` intermediate variables for absolute errors, this hint will be equivalently written:

```
delta -> delta1 + delta2;
```

2. Relative errors can be manipulated similarly. Given $\epsilon = \frac{x-MX}{MX}$, $\epsilon_1 = \frac{x-X}{X}$, and $\epsilon_2 = \frac{X-MX}{MX}$, the next hint may be used. In particular it is needed for the integration of polynomial approximation errors into the final error estimate.

```
epsilon -> epsilon1 + epsilon2
           + epsilon1 * epsilon2;
```

This is still a mathematical identity as one may check easily by developing the definitions.

3. When x is an approximation of MX and a relative error $\epsilon = \frac{x-MX}{MX}$ is known by the tool, x can be rewritten $MX \cdot (1+\epsilon)$. This kind of hint is useful in combination with the following one.

4. When manipulating fractional terms such as $\frac{Expr_1}{Expr_2}$ where $Expr_1$ and $Expr_2$ are correlated (for example one approximating the other), the interval division fails to give useful results if the interval for $Expr_2$ comes close to 0. In this case, one will try to write $Expr_1 = A \cdot Expr_3$ and $Expr_2 = A \cdot Expr_4$, so that the interval on $Expr_4$ does not come close to 0 anymore. The following hint is then appropriate:

```
Expr1 / Expr2 -> Expr3 / Expr4;
```

This rewriting rule is only valid if A is not zero, so the case $A = 0$ has to be handled separately.

All these hints are correct if both sides are mathematically equivalent. Gappa therefore checks this automatically. If the test fails - which is rare - it emits a warning to the user that he or she must review the hint by hand. Therefore, writing even complex hints is very safe: one may not introduce an error in the proof by writing a false hint without getting a warning.

However, finding the right hint that Gappa needs could be quite complex without completely mastering its theorem database and the algorithms used by its engine. Fortunately, a much simpler way is to build the proof incrementally and question the tool by adding and removing intermediate goals to prove, as the example in the following section shows.

5. EXTENDED EXAMPLE: A LOGARITHM

This section computes a relative error bound on the evaluation to a double-double of a polynomial approximating $\log(1+Z)$ where Z is a reduced argument. This computation is the core of the first step in `crlibm`'s current portable implementation of the natural logarithm [4]. The argument reduction used is errorless, but the reduced argument needs to be stored on a double-double, so $Z = z_h + z_l$. The proof that the argument reduction is exact is done by hand, and the reconstruction introduces no new difficulty (it merely consists in two successive double-double additions), so we do not show it here for the sake of brevity. For a full description of this implementation, including the second step, see [4].

5.1 Algorithm and C code

This polynomial evaluation inputs the double-double $Z = z_h + z_l$, and should return a double-double approximation to $p(Z)$. Evaluating the whole polynomial using double-double arithmetic would not be efficient: instead, the polynomial approximating $\log(1+Z)$ is written as follows:

$$p(Z) = Z - \frac{1}{2} \cdot Z^2 + Z^3 \cdot q(Z) \qquad (1)$$

and the respective terms are evaluated as follows: $\frac{1}{2} \cdot Z^2 = \frac{1}{2} \cdot (z_h + z_l)^2$ is approximated by $\frac{1}{2} \cdot z_h^2 + z_h z_l$, where z_h^2 is computed exactly as a double-double. Z^3 is approximated by z_h^3 computed in double precision, $q(Z)$ is a polynomial with double-precision coefficients, and is approximated by $q(z_h)$ so that it can be evaluated entirely in double.

The corresponding C code is given below.

```
1  q = c3 + zh * (c4 + zh * (c5 + zh * (c6 + zh *
       c7)));
2  Mul12(&zhSquareh, &zhSquarel, zh, zh);
3  zhCube = zh * zhSquareh;
4  polyUpper = zhCube * q;
5  zhSquareHalfh = zhSquareh * -0.5;
6  zhSquareHalfl = zhSquarel * -0.5;
7  zhzl = -1 * (zh * zl);
8  Add12(t1h, t1l, polyUpper, zhzl);
9  Add22(&t2h, &t2l, zh, zl, zhSquareHalfh,
       zhSquareHalfl);
10 Add22(&ph, &pl, t2h, t2l, t1h, t1l);
```

Here `Add12` (also known as Fast2Sum) is a sequence computing the exact sum of two doubles as a double-double. Similarly, `Mul12` computes the exact product of two doubles as a double-double. Finally, the procedure called `Add22` computes as a double-double the sum of two double-double numbers with a relative error less than 2^{-103} [5, 9].

The code is a typical example of floating-point code written for a target accuracy of about 2^{-62} (neglecting the lower significant argument z_l in all terms where it is not strictly needed, for instance). It also expresses some parallelism (line 1 and lines 2-3 can be evaluated concurrently). We have established the proof of the error of this code step by step as previously described [3].

5.2 Gappa error computation

Following the guidelines of Section 4, the C code can be translated into Gappa syntax as follows:

```
1 zh = float64ne(Z);
2 zl = Z - zh;
3 q float64ne= c3 + zh * (c4 + zh * (c5 + zh * (
      c6 + zh * c7)));
4 ZhSquarehl = zh * zh;
5 zhSquareh = float64ne(ZhSquarehl);
6 zhCube float64ne= zh * zhSquareh;
7 polyUpper float64ne= zhCube * q ;
8 ZhSquareHalfhl = -0.5 * ZhSquarehl;
```

```
9  zhzl = -1 * float64ne(zh * zl);
10 T1hl = polyUpper + zhzl;
11 T2hl = add_rel<103>(Z, ZhSquareHalfhl);
12 Phl = add_rel<103>(T2hl, T1hl);
```

Note that there is no rounding operator at line 8: we know that a multiplication by -0.5 is exact in IEEE-754-compliant arithmetic for the range of values we consider. The same holds for the multiplication by -1, line 9.

The next step is to express in Gappa what this code is intended to compute.

```
14 MQ = c3 + Z * (c4 + Z * (c5 + Z * (c6+Z*c7)));
15 MZSquare = Z * Z;
16 MZCube = Z * MZSquare;
17 MP = Z - 0.5*MZSquare + MZCube*MQ;
18 epsilon = (Phl - MLog1pZ) / MLog1pZ;
19 epsilonApproxPoly = (MP - MLog1pZ) / MLog1pZ;
```

Then we express the theorem to prove.

```
30 { (Z in [-1b-8,-1b-200] \/ Z in [1b-200,1b-8])
31  /\ |zl| in [1b-300,1]
32  /\ epsilonApproxPoly in [-1b-63,1b-63]
33   -> epsilon in [-1b-62,1b-62] }
```

Finally, some hints need to be written. This is the hardest part of the proof work, as they express all the optimizations and approximations the programmer has done when designing this numerical code. Unfortunately, space prevents us from showing in detail how they may be derived by interacting with Gappa (see [3]). Merely four more hints were needed for the given code.

The Gappa proof obtained is very concise: for our 10-line C code sequence that consists of 13 native double operations and 4 higher precision procedures, a Gappa file of about 100 lines is needed. Writing the Gappa file for the whole logarithm function was a matter of a few hours.

The tool computes the bounds in a few seconds on a recent machine and generates a formal proof for Coq of more than 4800 lines. If the proof had to be written in Coq by hand, it would probably require weeks of tedious work. Besides, most of this work would be lost if a part of the algorithm had to be rewritten. A Gappa description does not suffer from such a shortcoming: it can easily be adapted to a new implementation of the algorithm.

6. CONCLUSION AND PERSPECTIVES

Validating tight error bounds on the low-level, optimized floating-point code typical of elementary functions has always been a challenge, as many sources of errors cumulate their effect. Gappa is a high-level proof assistant that is well suited to this kind of proofs.

Using Gappa, it is easy to translate a part of a C program into a mathematical description of the operations involved with fair confidence that this translation is faithful. Expressing implicit mathematical knowledge one may have about the code and its context is also easy. Gappa uses interval arithmetic to manage the ranges and errors involved in numerical code. It handles most of the decorrelation problems automatically thanks to its built-in rewriting rules, and an engine which explores the possible rewriting of expressions to minimize the size of the intervals. If decorrelation remains, Gappa allows one to provide new rewriting rules, but checks them. All this is well founded on a library of theorems which allow the obtained computation to be translated to a proof checkable by a lower-level proof assistant such as Coq. Finally, the tool can be questioned during the process of building the proof so that this process may be conducted interactively.

Therefore, it is possible to get quickly a fully validated proof with good confidence that this proof indeed proves properties of the initial code. Gappa is by no means automatic: to apply it on a given piece of code requires exactly the same knowledge and cleverness a paper proof would. However, it requires much less work.

Gappa, at several stages of its development, has already been used to prove error bounds for parts of the logarithm, exponential, sine and tangent functions of the current `crlibm` distribution (0.11beta). Although its development is not over, the current version (0.5.4) is very stable and we may safely consider generalizing the use of this tool in the future developments of `crlibm`.

7. REFERENCES

[1] M. Daumas and G. Melquiond. Generating formally certified bounds on values and round-off errors. In *6th Conference on Real Numbers and Computers*, pages 55–70, 2004.

[2] F. de Dinechin, A. Ershov, and N. Gast. Towards the post-ultimate libm. In *17th Symposium on Computer Arithmetic*. IEEE Computer Society Press, June 2005.

[3] F. de Dinechin, Ch. Q. Lauter, and G. Melquiond. Assisted verification of elementary functions. Technical Report RR2005-43, LIP, September 2005.

[4] F. de Dinechin, Ch. Q. Lauter, and J.-M. Muller. Fast and correctly rounded logarithms in double-precision. Technical Report RR2005-37, LIP, September 2005. To appear in Theoretical Informatics and Applications.

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

[6] J. Harrison. Floating point verification in HOL light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory, 1997.

[7] W. Hofschuster and W. Krämer. FI_LIB, 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.

[8] D. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, 1973.

[9] Ch. Q. Lauter. Basic building blocks for a triple-double intermediate format. Technical Report RR2005-38, LIP, September 2005.

[10] G. Melquiond and S. Pion. Formal certification of arithmetic filters for geometric predicates. In *Proceedings of the 15th IMACS World Congress on Computational and Applied Mathematics*, 2005.

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

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