Toward the Integration of Numerical Computations into the OMSCS Framework

Jacques Calmet[1] and Vincent Lefèvre[2⋆]

[1] Universität Karlsruhe
Fakultät für Informatik
IAKS Prof. Calmet
D-76128 Karlsruhe, Germany
calmet@ira.uka.de
[2] INRIA Lorraine
615 rue du Jardin Botanique
54602 Villers-lès-Nancy Cedex, France
vincent@vinc17.org

Abstract. Computer algebra systems and automated theorem provers, which have complementary abilities, can be integrated to form an Open Mechanized Symbolic Computation System (OMSCS). This framework could be extended to integrate numerical computation systems. This paper aims at showing what problems can occur when dealing with numerical computations and what can be done to solve them or at least to provide a clear meaning of a numerical result; it constitutes a step toward this integration.

1 Introduction

In [1], P. G. Bertoli, J. Calmet, F. Giunchiglia and K. Homann introduced a general framework integrating computer algebra systems and automated theorem provers, named OMSCS (Open Mechanized Symbolic Computation System) and showed how this integrated system can be used to solve problems which could not be tackled by each single system alone. Numerical computations involving real or complex numbers and performed in a floating-point arithmetic, a fixed-point arithmetic or any other arithmetic could be integrated into this framework (in this paper, we will focus on floating-point arithmetic). But contrary to systems that operate on exact data, numerical systems need to perform approximations; exact mathematical results are not even representable in general. This is a real problem for the integration with other systems, as this paper will show. The solution of doing exact computations or using computer algebra (working on exact data) is too slow in general, or even often impossible. Thus one needs to take approximations into account, and properties of a numerical software may be very different, depending on how these approximations are done. This can be summarized as follows. In addition to fast computations, one would want:

- accurate and well-specified results: the absolute value of the difference between the returned result and the mathematical result must be bounded above by some value (which could be provided either by the user or by the computation system);
- consistent results, i.e. preserved mathematical properties (monotonicity, identities, etc);
- reproducible computations, i.e. unicity of the results, across different architectures or different programs (that could be different versions of a same software).

These goals have often been dealt with for the arithmetic operations in floating-point arithmetic (specified by the IEEE-754 standard), and more generally for conventional mathematical functions, such as the exponential, the logarithm, the trigonometric functions, and so on; but they also extend to whole programs (instead of libraries of given elementary mathematical functions) when considering software integration, where some parts are seen as black boxes.

In the following, we will focus on what can be done concerning the results and their interpretation; we will neither go into detail on how this can be done, nor deal with the complexity (except when this is a real problem).

⋆ Work partially supported by the Calculemus Research Training Network HPRN_CT-2000-00102.

Section 2 introduces general problems that occur in numerical computations, related to the above requirements. Section 3 presents a general way (without details) to formalize approximations and work with them (accuracy goal). Section 4 deals with the unicity goal. Section 5 is a discussion on proofs on numerical computations.

2 General Problems

2.1 Continuity Problems

A program can be seen as a sequence of operations, where some of them can be regarded as follows: such an operation depending on entries x_1, x_2, ..., x_k consists in returning an approximation $\tilde{f}(x_1, \ldots, x_k)$ to the exact result $f(x_1, \ldots, x_k)$ of a mathematical function f whose range is the real numbers[1]. How these operations are linked together is not our concern; we do not want to restrict to some kind of language, for instance. The whole program also consists in computing an approximation to the result of a mathematical function.

Note that some algorithms may use the fact that an operation returns a rounded result in the working precision. In this case, such an operation is not regarded here as returning an approximation. It just corresponds to an exact mathematical function (that depends on the working precision). More is said about that later (Section 3).

Consider a program (or a part of a program) that returns a numerical value. This is an approximation to the exact mathematical result with some error bounded by 2^{-m}, where m is an integer called the *accuracy*[2] (in bits). A way to improve the accuracy is to increase the working precision. If all the basic functions are continuous and the program corresponds to a fixed composition of these functions, then by increasing the working precision, one can obtain a result with an error less than 2^{-m} for any m, i.e. a result as accurate as one wants. If there is some form of discontinuity in the program, getting a potentially infinite accuracy by increasing the precision is not guaranteed.

Before going further concerning the discontinuity problems, let us assume that we are in the easy case: we can get results as accurate as we want. The accuracy goal can be fulfilled, but what about the unicity goal? This is particularly important to get the same results in different contexts (e.g. different architectures, after an internal optimization, etc). For a given target accuracy, an infinite number of values are valid (more precisely, a whole interval). Instead, the user may provide a target precision[3] (say, the least significant bit has a weight 2^{-p}) and the most accurate result in this precision is required; halfway cases must be fully specified, for instance by requiring the bit of position $-p$ of the result to be 0 in such cases. This requirement is called *exact rounding*. Exact rounding is more often considered with basic functions, such as the arithmetic operations and the square root in the IEEE-754 standard [9], or the *elementary functions* in some mathematical libraries (for instance, IBM's MathLib[4] and Crlibm[5] from the Arenaire[6] project in the IEEE-754 double precision, MPFR[7] in multiple precision), possibly in *directed* rounding modes (rounding toward $-\infty$, toward $+\infty$ or toward 0).

If the error bound of an approximation is less than the distance between the computed value and the middle of two consecutive representable values in the target precision, called a *breakpoint*, then rounding this computed value is equivalent to rounding the exact value (as if it were computed with an infinite precision): the exactly rounded result can be returned. Otherwise it is not possible to decide what value should be returned; this problem is called the *Table Maker's Dilemma* (TMD, for short). In this case, more internal precision is necessary and one can restart the computations

[1] We could also deal with complex numbers or matrices of real or complex numbers, with a specific representation and norm. But for the sake of simplicity, we restrict to real numbers here.

[2] This is here the *absolute* accuracy. We may also consider the relative accuracy or the accuracy on the mantissa in a floating-point arithmetic. But the problems are basically the same, except for 0, where only the absolute accuracy is defined.

[3] In general, this would be a mantissa size, but for consistency, we consider the absolute precision here.

[4] http://oss.software.ibm.com/mathlib/

[5] http://perso.ens-lyon.fr/catherine.daramy/crlibm.html

[6] http://www.ens-lyon.fr/LIP/Arenaire/

[7] http://www.mpfr.org/

with more precision [39]. Therefore, assuming that one can compute an accurate enough result, providing an exactly rounded result is possible, unless the exact value is a breakpoint, in which case the TMD occurs with any internal precision and only a mathematical proof would solve it (or if one can make sure that all the internal computations are exact, but this is very rare). But one can notice that this is again a continuity problem: indeed, the breakpoints are the discontinuity points of the *round* function.

One should note that choosing a directed rounding mode instead of the rounding to the nearest value or some other similar solution for unicity would not solve the general problem. The breakpoints would be different, but there would always be breakpoints. Deciding whether or not the exact result of a computation is a breakpoint is equivalent to the well-known zero recognition problem, i.e. decide whether a result is zero (more is said on this in Section 4).

For more information on the TMD concerning the elementary functions, see [23, 16–18, 13–15, 34].

2.2 Cancellation of the Rounding Errors

In general, rounding errors partially compensate each other in a computation. But in proofs (static error analysis) or in interval arithmetic (with outward rounding, of course), the effects of rounding errors must be added, except in particular cases (for instance, Newton's iteration to solve an equation is self-corrective). Thus one can obtain very pessimistic error bounds, at least in the average case. However, this is not our primary concern here. It is still interesting to detect exact results, though, to avoid generic rounding error bounds in such cases.

2.3 Validity and Consistency

Because of rounding errors, some mathematical properties, like

$$\frac{|X|}{\sqrt{X^2+Y^2}} \leq 1$$

(corresponding to a cosine) for real numbers X and Y (not both equal to zero) or the monotonicity of a computed function are no longer necessarily satisfied. As a consequence, one may obtain undefined results when a computation leads to an argument that is outside the domain of some function, whereas if the computations were exact (or carried out with a higher precision), such a problem would not have occurred. For instance, consider the arc cosine of the above expression. A computation model should take care of that.

Similarly, rounding errors may lead to inconsistencies. For instance, in computational geometry, unless special care is taken concerning the accuracy, contradictory predicates may both hold, like the following ones: points A, B, C are aligned, but points A, C, B are not aligned (using a different permutation in the alignment formula).

Exact rounding on the final result allows to ensure validity and consistency, but this is not always possible. Solving consistency problems in computational geometry is particularly difficult as they are often linked to discontinuities (such as the alignment condition) and the problems described in Section 2.1.

3 Toward a Universal Model

An error analysis can generally be described as follows: for each considered number, one has an approximation and an error bound, with possible variants. For instance, the error bound may be strict or loose; the sign may be known; instead of an approximation and an error bound, one may have a lower bound and an upper bound (mainly in interval arithmetic).

To ease proofs and interoperability, it would be interesting to have a universal model that can express all the above variants and possibly others. An object would be a set of attributes associated with a number in some computation. All these attributes do not need to be defined. These attributes can include:

- The exact value x. Of course, as soon as an approximation occurs, this value is unknown (therefore not defined).
- An approximated value v (say, the best known approximated value).
- An absolute bound E on the error (useful for proofs).
- A relative bound on the error.
- A lower bound $x_{\inf}$, possibly minus infinity (useful for interval arithmetic).
- An upper bound $x_{\sup}$, possibly plus infinity (useful for interval arithmetic).
- Validity flag. Such information is useful when an algorithm has not been proved to be correctly defined (in the mathematical sense) on all entries. In other words, the set of the real numbers $\mathbb{R}$ is extended with a special value NaN (known as *not a number*[8]).
- Information about the error bounds: loose or strict bounds.

Values are expressed in some system (this may be floating point, fixed point, rational, and so on), possibly a symbolic one when this makes sense. For instance, a bound may be expressed as a function of the internal precision by a static analysis.

Objects could also be provided with an additional attribute allowing to determine the condition number (that would be more or less equivalent to the computation of E, where the input object has an error ε and as if internal computations were performed in an infinite precision, i.e. without taking rounding errors into account).

Some attributes may be deduced from other attributes. For instance, from a lower bound and an upper bound, one can compute a centered approximated value and a corresponding error bound. Conversely, from an approximated value and an error bound, one can compute a lower bound and an upper bound. To go further, one can say that in the initial state, no attributes are defined. Then a computation consists in increasing knowledge by defining attributes, just like an error analysis. Deductions can be done using general properties like $|x - v| \leq E$ and other formulas.

The objects could be extended to more complicated ones such as complex numbers or vectors, allowing to take into account correlated errors and the structure of such objects.

Some algorithms are based on the exact rounding (or a weaker rounding behavior, such as a faithful rounding) and in general, would not work if operations were exact. This is the case of algorithms working on floating-point expansions to perform arbitrary precision computations [5, 26, 32, 33, 7]. The rounding is here regarded as a feature. For instance, if for an addition in the algorithm, the rounding is important, then the associated exact mathematical operation will not be the conventional addition over the real numbers, but the *rounded* addition. This rounded operation is formalized by conventional mathematical operations, e.g.

$$\frac{\lfloor s \times 2^{n-1} \rfloor}{2^{n-1}}$$

as done in [22].

What has been described until now allows to track the loss of accuracy and to do some proofs on the results, e.g. to prove that the exact result lies in some interval. We have dealt with values, but not much with operators. An operator is not just a mathematical function. In practice, a function evaluation is also performed with some precision. Information about how a function evaluation is performed could be provided to be able to deduce properties of the computed results, e.g. error bounds. For operators that work on approximated values, such information can be the precision, with exact rounding or not, with faithful rounding or not, and so on. For operators that work on $x_{\inf}$ and $x_{\sup}$ (like interval arithmetic), this can be properties of the interval bounds.

How objects are modified and interact with each other must be defined with care. Let us take an example showing the difficulty: the iRRAM package[9] (or some system behaving in a similar way), which aims to implement a Real-RAM model. A simple computation can be regarded as a DAG (directed acyclic graph), the nodes corresponding to objects. With some conventional software, each object would be computed once (possibly with several passes, e.g. one corresponding to a

[8] In IEEE arithmetic, the special NaN value is also used to mean that nothing is known about a real value. This is not the case here.

[9] http://www.informatik.uni-trier.de/iRRAM/

compiler optimization and one corresponding to the running program[10]). But with iRRAM, if the package detects at some point in the program that early computations were not carried out with sufficient precision, the internal precision is increased and the DAG is recomputed. That is, if we want to see the iRRAM package as a black box, objects will magically be modified.

4 Returning a Unique Result

A model based on the Section 3 remarks will allow to lead to completely specified results if sufficient information is provided concerning the operators. That would be the case on true IEEE-754 systems with some constraints: computations are performed in double precision, only the operations defined by the IEEE-754 standard are used ($+$, $-$, $\times$, $/$, $\sqrt{}$), and optimizations modifying the results (like using the FMA[11] operation on processors that have one) are avoided. In most cases, though there exist standards, unicity is not guaranteed without writing special code to ensure it. For instance, the IEEE-754 standard allows intermediate computations to be performed in an extended precision. The ISO/IEC 9899:1999 standard (C language) has even fewer requirements.

Also one should note that unicity at the low level (basic functions) does not replace an error analysis. Without an error bound or an enclosing interval, the final result may be completely meaningless. As an example, let us consider the following sequence by J.-M. Muller:

$$\begin{cases} u_0 &= 2 \\ u_1 &= -4 \\ u_{n+1} &= 111 - \dfrac{1130}{u_n} + \dfrac{3000}{u_n u_{n-1}}. \end{cases}$$

One can prove that u_n converges to 6, but on any machine (not using interval arithmetic or anything similar), u_n seems to converge to 100 very quickly.

Ensuring exact rounding in some precision at the high level instead of the low level would have important advantages:

- Final accuracy would be guaranteed.
- Results would not depend on the internal algorithms and how code has been compiled. So, new, better algorithms could be used, and code could be optimized further by a compiler, having fewer constraints on the intermediate computations.

But can this be done and how difficult is it? As it has been said in Section 2.1, the main problem is related to discontinuities (we recall that this is equivalent to the zero recognition problem, also called the *constant problem* or the *identity problem*).

First, the general case is undecidable [27]. However, particular classes of problems have been studied, in particular by D. Richardson [28, 29, 12, 30] and J. van der Hoeven [35, 31, 36, 37]. But for the largest classes, results are based on conjectures and/or the zero test is very complex (and takes many resources, both time and memory). [30] deals with polynomials and the zero test is theoretical (probabilistic and depending on an unknown constant); [28] introduces some exponentials, but the zero test is based on Schanuel's conjecture and is very complex. [29] and [12] deal with the *Uniformity Conjecture* (based on probabilistic arguments, and very pessimistic on most cases), which, if true, allows to design a very simple zero test though taking resources, and J. van der Hoeven defined similar conjectures, called *witness conjectures*, in [35], but he eventually found counter-examples to the weakest witness conjectures and the Uniformity Conjecture [37]. Concerning the TMD, T. Lang and J.-M. Muller give bounds for algebraic functions in [11].

Despite the above remarks, in practice, a zero test is simple for most cases, as for these cases, computing with enough internal precision would be sufficient[12]. Problems occur only in particular cases (possibly rare or even impossible in many applications); depending on the context, one could return some form of error to warn the user so that he can take appropriate measures.

[10] Well, refinements could be allowed, in which case that would be a behavior similar to iRRAM.

[11] Fused multiply-add.

[12] This may take a lot of memory (more precisely, with an unbounded complexity): consider $(2^n + 2^{-n}) - 2^n$ with large n, for instance.

But solving this problem is particularly useful in geometric computations [38], where such particular cases are common (due to intrinsic alignments, other general geometric properties and degenerate inputs). C. Yap et al. wrote a package for this purpose: *Real/Expr*, and now the *Core* library [6, 25, 10], computing radical expressions, and where the zero test is based on bounds for such expressions [3, 4, 19–21].

5 Proofs and Standards

To be able to prove a program, the underlying system must have strong enough specifications, that can be given by a standard (e.g. ISO/IEC 9899 for the C language, IEEE 754 for floating-point arithmetic operations). A common difficulty is to correctly interpret those specifications and give a correct formalization for proofs. Also, the underlying system is not necessarily proved and in particular, it may have bugs. But concerning the numerical domain, many other difficulties arise in practice.

For instance, let us consider a program written in C, doing numerical computations using the type `double` (this is the floating-point type in C that provides the most portability).

First, if one considers the C standard only, then almost nothing is specified, in particular concerning the accuracy. Before systems supported the IEEE-754 standard, one could not even expect sensible results; for instance, on some Crays, 14.0/7.0 does not give 2 exactly, but a result a little smaller (as a consequence, in C, `(int) (14.0/7.0)` would give 1)[13], and there are machine numbers x such that $1 \times x$ gives an overflow (this may still occur with IEEE-754 arithmetic, see below). Even if the processor conforms to the IEEE-754 standard, this does not mean that a C implementation will; indeed special processor instructions, such as the FMA on the PowerPC, could be used.

Nowadays the IEEE-754 standard is generally supported, but proofs based on this standard must take all the points into account. This includes the subnormals, of course, but also the fact that intermediate results may be computed in extended precision (this behavior is also allowed by the C standard); this is the case with Linux on x86-compatible processors. One should note that a proof valid for any precision would not necessarily apply here since implicit (and uncontrolled) conversions from extended to double precision may introduce unwanted behavior; in particular, such conversions may generate overflows, as the exponent range is also larger than in double precision on x86. More information can be found in the addendum *Differences Among IEEE 754 Implementations* of [8]. Some bugs in language implementations (in the sense that they do not conform to the specifications) are related to the extended precision: in the *gcc* compiler (which does not convert results to the destination type in casts and assignments), in Java Virtual Machines and in XPath implementations (where double-precision computations are required).

The elementary mathematical functions (exponential, logarithm, trigonometric functions, etc) are not covered by the IEEE-754 standard. Thus one cannot assume that they are computed with some error bound on any architecture. Worse, implementations are still deficient. For instance, in directed rounding modes, the result is not always rounded in the correct direction (which precludes interval arithmetic) and some mathematical libraries may give completely wrong results, if not crash. Another example is the Intel Pentium processor sine and cosine functions not following Intel's own specifications: the rounding error may be much higher than the one claimed by Intel due to a loss of accuracy in the range reduction.[14]

Another point is that transcriptions to a proof checker that appear to be direct may hide bugs due to subtle differences between the language specifications and what humans may intuitively interpret. For instance, the floating-point expressions `x == y` and `x - y == 0` are not equivalent (due to infinities in the IEEE-754 standard). The C language also has implicit conversions that must be taken into account in proofs. This is particularly visible on integer arithmetic when mixing signed and unsigned integer types. For instance, the following function:

[13] The original example was the Fortran instruction `I = 14.0/7.0`, which gives the result 1.
[14] http://www.naturalbridge.com/floatingpoint/intelfp.html

```
long long umd(void)
{
  long a = 1;
  unsigned int b = 2;
  return a - b;
}
```

may return -1, return a positive value (e.g. $2^{32}-1$) or have an undefined behavior, depending on the implementation. Y. Bertot, N. Magaud and P. Zimmermann "proved" the implementation of the GMP square root in C (i.e. more than just the algorithm), using the Coq proof assistant [2]; unfortunately type checking, implicit conversions in the C integer arithmetic and integer overflow checking are not mentioned in the paper, and the `mpn_dq_sqrtrem` function, one of the proved functions, contains a bug[15] a bit similar to what happens in the above code (though this bug does not show up with current C compilers, where signed arithmetic is implemented like a modular arithmetic).

More generally, it is important to be aware of the completeness or the incompleteness of the proofs. For instance, the following fact is mentioned in an Intel report [24]: the discovery of a bug in the Pentium Pro floating-point execution unit disturbed the authors because this unit had been subject of a previous formal verification project.

6 Conclusion

This paper described the problems linked to numerical computations, concerning the accuracy and the unicity of the results, and started to define a general model for such computations. Future work should consist in an implementation and applying examples (that would also help to improve the model). Then the objects could be extended to complex numbers and matrices, for instance.

References

1. P. G. Bertoli, J. Calmet, F. Giunchiglia, and K. Homann. Specification and integration of theorem provers and computer algebra systems. In J. Calmet and J. Plaza, editors, *Proceedings of the International Conference on Artificial Intelligence and Symbolic Computation (AISC-98)*, volume 1476 of *LNAI*, pages 94–106, Berlin, September 1998. Springer.
2. Y. Bertot, N. Magaud, and P. Zimmermann. A proof of GMP square root. *Journal of Automated Reasoning*, 29(3–4):225–252, December 2002. Special Issue on Automating and Mechanising Mathematics: In honour of N.G. de Bruijn.
3. C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. A strong and easily computable separation bound for arithmetic expressions involving radicals. *Algorithmica*, 27:87–99, 2000.
4. C. Burnikel, S. Funke, K. Mehlhorn, S. Schirra, and S. Schmitt. A separation bound for real algebraic expressions. *Lecture Notes in Computer Science*, 2161, 2001.
5. T. J. Dekker. A floating-point technique for extending the available precision. *Numer. Math.*, 18:224–242, 1971.
6. T. Dubé, K. Ouchi, and C. Yap. Tutorial for `Real/Expr` package, 1997. *http://cs.nyu.edu/exact/realexpr/tutorial.ps.gz*
7. C. Finot-Moreau. *Preuves et algorithmes utilisant l'arithmétique flottante normalisée IEEE.* PhD thesis, École Normale Supérieure de Lyon, Lyon, France, July 2001. *ftp://ftp.ens-lyon.fr/pub/LIP/Rapports/PhD/PhD2001/PhD2001-03.ps.Z*
8. D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, March 1991. An edited reprint is available at http://cch.loria.fr/documentation/IEEE754/ACM/goldbergSUN.ps from Sun's Numerical Computation Guide; it contains an addendum *Differences Among IEEE 754 Implementations*, also available at http://cch.loria.fr/documentation/IEEE754/ACM/addendum.html.
9. IEEE standard for binary floating-point arithmetic. Technical Report ANSI-IEEE Standard 754-1985, New York, 1985. Approved March 21, 1985: IEEE Standards Board, approved July 26, 1985: American National Standards Institute, 18 pages.

[15] Page 16 of the paper, line 22.

10. V. Karamcheti, C. Li, L. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *Proceedings of the Conference on Computational Geometry (SCG'99)*, pages 351–359, New York, June 1999. ACM Press.
http://cs.nyu.edu/exact/doc/core.pdf.gz
11. T. Lang and J.-M. Muller. Bounds on runs of zeros and ones for algebraic functions. In N. Burgess and L. Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pages 13–20, Vail, Colorado, USA, 2001. IEEE Computer Society Press, Los Alamitos, CA.
http://www.ece.ucdavis.edu/acsel/arithmetic/arith15/papers/ARITH15_Lang.pdf
12. S. Langley and D. Richardson. What can we do with a solution? In V. Brattka, M. Schröder, and K. Weihrauch, editors, *CCA 2002 Computability and Complexity in Analysis*, volume 66 of *Electronic Notes in Theoretical Computer Science*, Amsterdam, 2002. Elsevier. 5th International Workshop, CCA 2002, Málaga, Spain, July 12–13, 2002.
13. V. Lefèvre. *Moyens arithmétiques pour un calcul fiable*. PhD thesis, École Normale Supérieure de Lyon, Lyon, France, January 2000.
http://www.vinc17.org/research/papers/these.ps.gz
14. V. Lefèvre and J.-M. Muller. Worst cases for correct rounding of the elementary functions in double precision. Research report RR2000-35, Laboratoire de l'Informatique du Parallélisme, Lyon, France, 2000.
http://www.vinc17.org/research/papers/rr_worstcases.ps.gz
15. V. Lefèvre and J.-M. Muller. Worst cases for correct rounding of the elementary functions in double precision. In N. Burgess and L. Ciminiera, editors, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pages 111–118, Vail, Colorado, 2001. IEEE Computer Society Press, Los Alamitos, CA.
http://www.ece.ucdavis.edu/acsel/arithmetic/arith15/papers/ARITH15_Lefevre.pdf
16. V. Lefèvre, J.-M. Muller, and A. Tisserand. Towards correctly rounded transcendentals. In *Proceedings of the 13th IEEE Symposium on Computer Arithmetic*, Asilomar, USA, 1997. IEEE Computer Society Press, Los Alamitos, CA.
http://www.ece.ucdavis.edu/acsel/arithmetic/arith13/papers/ARITH13_Lefevre.pdf
17. V. Lefèvre, J.-M. Muller, and A. Tisserand. The table maker's dilemma. Research report RR1998-12, Laboratoire de l'Informatique du Parallélisme, Lyon, France, 1998.
http://www.vinc17.org/research/papers/rr_tmd98.ps.gz
18. V. Lefèvre, J.-M. Muller, and A. Tisserand. Towards correctly rounded transcendentals. *IEEE Transactions on Computers*, 47(11):1235–1243, November 1998.
http://perso.ens-lyon.fr/jean-michel.muller/Nov98.pdf
19. C. Li. *Exact Geometric Computation: Theory and Applications*. PhD thesis, New York University, January 2001.
http://cs.nyu.edu/exact/doc/chenliThesis.ps.gz
20. C. Li and C. Yap. A new constructive root bound for algebraic expressions. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-01)*, pages 496–505, New York, January 2001. ACM Press. Extended abstract at http://cs.nyu.edu/exact/doc/rootBd_abs.ps.gz.
21. C. Li and C. Yap. Recent progress in exact geometric computation, June 2001. Paper based on a talk presented at the DIMACS Workshop on Algorithmic and Quantitative Aspects of Real Algebraic Geometry in Mathematics and Computer Science, March 12–16, 2001.
http://cs.nyu.edu/exact/doc/dimacs.ps.gz
22. J. S. Moore, T. Lynch, and M. Kaufmann. A mechanically checked proof of the AMD5K86 floating-point division program. *IEEE Transactions on Computers*, 47(9):913–926, September 1998. Early draft available at http://www.cli.com/news/divide.ps and http://www.cs.utexas.edu/users/moore/publications/divide_paper.ps.gz.
http://dlib.computer.org/tc/books/tc1998/pdf/t0913.pdf
23. J.-M. Muller and A. Tisserand. Towards exact rounding of the elementary functions. In G. Alefeld, A. Frommer, and B. Lang, editors, *Scientific Computing and Validated Numerics (proceedings of SCAN'95)*. Akademie Verlag, 1996.
24. J. O'Leary, X. Zhao, R. Gerth, and C.-J. H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, 3(Q1), February 1999.
http://www.intel.com/technology/itj/q11999/pdf/floating_point.pdf
25. K. Ouchi. `Real/Expr`: Implementation of an exact computation package. Master's thesis, New York University, January 1997.
http://cs.nyu.edu/exact/doc/KoujiThesis.ps.gz
26. D. M. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. Matula, editors, *Proceedings of the 10th IEEE Symposium on Computer Arithmetic*, pages 132–144, Grenoble, France, June 1991. IEEE Computer Society Press, Los Alamitos, CA.
http://www.ece.ucdavis.edu/acsel/arithmetic/arith10/papers/ARITH10_Priest.pdf

27. D. Richardson. Some unsolvable problems involving elementary functions of a real variable. *Journal of Symbolic Logic*, (33):514–520, 1968.
28. D. Richardson. A simplified method of recognizing zero among elementary constants. In A. H. M. Levelt, editor, *ISSAC'95: Proceedings of the 1995 International Symposium on Symbolic and Algebraic Computation: July 10–12, 1995, Montreal, Canada*, pages 104–109, New York, NY 10036, USA, 1995. ACM Press.
http://www.acm.org/pubs/citations/proceedings/issac/220346/p104-richardson/
29. D. Richardson. The uniformity conjecture. In J. Blanck, V. Brattka, and P. Hertling, editors, *Computability and Complexity in Analysis*, volume 2064 of *Lecture Notes in Computer Science*, pages 253–272, Berlin, 2001. Springer. 4th International Workshop, CCA 2000, Swansea, UK, September 2000.
http://link.springer-ny.com/link/service/series/0558/papers/2064/20640253.pdf
30. D. Richardson and A. El-Sonbaty. Use of algebraically independent numbers for zero recognition of polynomial terms. April 2003.
http://www.bath.ac.uk/~masdr/shortest.ps
31. J. R. Shackell and J. van der Hoeven. Complexity bounds for zero-test algorithms. Technical Report 2001-63, Prépublications d'Orsay, 2001.
http://www.math.u-psud.fr/~vdhoeven/Publs/2001/zerotest.ps.gz
32. J. R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proceedings of the 12th Annual ACM Symposium on Computational Geometry*, pages 141–150, Philadelphia, Pennsylvania, 1996.
http://www.cs.cmu.edu/afs/cs/project/quake/public/papers/robust-predicates.ps
33. J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. In *Discrete and Computational Geometry*, volume 18, pages 305–363, 1997.
http://www.cs.cmu.edu/afs/cs/project/quake/public/papers/robust-arithmetic.ps
34. D. Stehlé, V. Lefèvre, and P. Zimmermann. Worst cases and lattice reduction. In J.-C. Bajard and M. Schulte, editors, *Proceedings of the 16th IEEE Symposium on Computer Arithmetic*, pages 142–147, Santiago de Compostela, Spain, 2003. IEEE Computer Society Press, Los Alamitos, CA.
http://www.ece.ucdavis.edu/acsel/arithmetic/arith16/papers/ARITH16_Stehle.pdf
35. J. van der Hoeven. Zero-testing, witness conjectures and differential diophantine approximation. Technical Report 2001-62, Prépublications d'Orsay, 2001.
http://www.math.u-psud.fr/~vdhoeven/Publs/2001/witconj.ps.gz
36. J. van der Hoeven. A new zero-test for formal power series. In T. Mora, editor, *Proc. ISSAC '02*, pages 117–122, Lille, France, July 2002.
http://www.math.u-psud.fr/~vdhoeven/Publs/2002/issac02.ps.gz
37. J. van der Hoeven. Counterexamples to witness conjectures. Technical Report 2003-43, Prépublications d'Orsay, 2003.
http://www.math.u-psud.fr/~vdhoeven/Publs/2003/cexconj.ps.gz
38. C. Yap. Robust geometric computation. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 41. CRC Press, 2003.
http://cs.nyu.edu/exact/doc/handbook03.ps.gz
39. A. Ziv. Fast evaluation of elementary mathematical functions with correctly rounded last bit. *ACM Transactions on Mathematical Software*, 17(3):410–423, September 1991.