Motivations for an arbitrary precision interval arithmetic and the MPFI library

Nathalie Revol, Fabrice Rouillier

Laboratoire de l'Informatique du Parallélisme

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

CNRS CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE

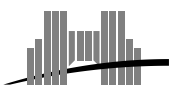
Motivations for an arbitrary precision interval arithmetic and the MPFI library

N. Revol[1] and F. Rouillier[2]

July 2002

Research Report N° 2002-27

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



INRIA

Motivations for an arbitrary precision interval arithmetic and the MPFI library

N. Revol[3] and F. Rouillier[4]

July 2002

Abstract

This paper justifies why an arbitrary precision interval arithmetic is needed: to provide accurate results, interval computations require small input intervals; this explains why bisection is so often employed in interval algorithms.
The MPFI library has been built in order to fulfill this need: indeed, no existing library met the required specifications. The main features of this library are briefly given and a comparison with a fixed-precision interval arithmetic, on a specific problem, is presented: it shows that the overhead due to the multiple precision is completely admissible.
Eventually, some applications based on MPFI are given: robotics, isolation of polynomial real roots (by an algorithm combining symbolic and numerical computations) and approximation of real roots with arbitrary accuracy.

Keywords: arbitrary precision interval arithmetic, reliability and accuracy

Résumé

Cet article justifie le besoin d'une arithmétique par intervalles en précision arbitraire : pour fournir des résultats précis, un calcul par intervalles requiert des intervalles en entrée qui soient fins ; c'est pour cette raison que la bissection est un procédé si souvent employé dans les algorithmes par intervalles.
La bibliothèque MPFI a été développée pour répondre à ce besoin : en effet, aucune bibliothèque existante n'offrait de spécifications satisfaisantes. Les caractéristiques de cette bibliothèque sont rapidement données puis une comparaison avec une bibliothèque d'arithmétique par intervalles en précision fixée est menée sur un problème spécifique : elle met en évidence le fait que le surcoût lié à la gestion de la précision multiple est tout à fait acceptable.
Pour terminer, quelques applications basées sur MPFI sont présentées : robotique, isolation des racines réelles de polynômes (par un algorithme combinant calcul symbolique et calcul numérique) et approximation avec une précision arbitraire de zéros réels.

Mots-clés: arithmétique par intervalles en précision arbitraire, calcul garanti et précis

Motivations for an arbitrary precision interval arithmetic and the MPFI library

N. Revol[¶] and F. Rouillier[‖]

July 2002

Motivations for changing arithmetic

Nowadays, computations involve more and more operations and consequently errors. The limits of applicability of some numerical algorithms are now reached: for instance the theoretical stability of a dense matrix factorization (LU or QR) is ensured under the assumption that $n^3u < 1$, where n is the dimension of the matrix and $u = 1^+ - 1$, with 1^+ the smallest floating-point larger than 1; this means that n must be less than 200,000, which is almost reached by modern simulations. The numerical quality of solvers is now an issue, and not only their mathematical quality. Let us cite studies performed by the CEA (French Nuclear Agency) on the simulation of nuclear plant accidents and also softwares controlling and possibly correcting numerical programs, such as Cadna [8] or Cena [24].

Another approach consists in computing with certified enclosures, namely interval arithmetic [27, 2, 21]. The fundamental principle of this arithmetic consists in replacing every number by an interval enclosing it. For instance, π cannot be exactly represented using a binary or decimal arithmetic, but it is certified that π belongs to $[3.14159, 3.14160]$. The advantages of interval arithmetic are numerous. On the one hand, it exhibits the property of *validated* or *certified computing*. On the other hand, computer implementations are based on outward roundings and thus computed results take into account rounding errors and constitute a way to estimate these errors. A last and very important advantage, even if it is often less known, is that this arithmetic provides global information: for instance, it provides the range of a function over a whole set S, which is crucial for global optimization; furthermore, if this range is a subset of S, then Brouwer's theorem states that this function has a fixed-point and this can be used by Newton's algorithm for instance. Such properties cannot be reached without set computing, and interval arithmetic computes with sets and is easily available.

However, in spite of the improvements in interval analysis, the problem of overestimation, *i.e.* of enclosures which are far too large and thus inaccurate,

[¶]Lab. ANO, University of Lille and CNRS/ENSL/INRIA project Arenaire, LIP, École Normale Supérieure de Lyon, France, `Nathalie.Revol@ens-lyon.fr`

[‖]Project Spaces, LORIA/INRIA/LIP 6, France, `Fabrice.Rouillier@loria.fr`

seems to be the destiny of interval computation when it is implemented using fixed-precision floating-point arithmetic. Computing with intervals provides guaranteed results, but the bounds can be far apart even when the input data are provided with the machine precision; a remedy for this phenomenon consists in computing with a higher precision. This proposal is the core of the MPFI library (*Multiple Precision Floating-point Interval arithmetic library*), a library implementing arbitrary precision interval arithmetic which is described in this paper.

This quest for extra accuracy can be found in other works such as [7] where polynomial expressions are symbolically rewritten before being evaluated, so as to reduce the overestimation due to dependency, or by [3] where high-order Taylor expansions are used. In this latter work, the time overhead is about 1500 for a single function evaluation, however it is compensated by the reduction in the number of steps performed by the algorithm. Real-world applications where extra accuracy is required are to be found in automatics (we have been asked to integrate linear systems with high accuracy) or chemistry: determining a molecular conformation entails the minimization of an energy function and requires accurate evaluations of this energy function.

Most usual interval arithmetic libraries (Profil/BIAS [23], Intlib [17]...) or compilers (Sun Forte, *e.g.* C++ [35]) are based on *double* floating-point numbers and do not propose arbitrary precision interval arithmetic. The XSC languages [21] include a long accumulator type for accurate dot product, which has also been used for Horner evaluation of a polynomial in the interval Newton algorithm [22]. However, this long accumulator type is not intended for a general use throughout the computation, and for instance it is not possible to compute a division or a logarithm using long accumulators. Furthermore, the maximal precision is limited, since the long accumulator type relies on the double type, whose range of exponents is $\{-1022, \cdots 1023\}$. Several multiple precision interval packages are available. Let us quote for instance intpak [10] and intpakX [12] for Maple or a similar package for Mathematica [20]. Due to unverified assumptions on the roundings of elementary functions (0.6 ulp for intpak in Maple, 1 ulp for Mathematica), to bugged roundings (for instance, with 3 decimal digits, the rounding towards 0 of $1 - 9.10^{-5}$ gives 1 instead of 0.999 in Maple v6 and v7), and to several other undue assumptions, these packages cannot be considered as reliable. Other works include the "range arithmetic" [1] and IntLab [33]. *range* is a multiple precision library which aims at indicating the number of correct digits rather than at performing interval arithmetic, and which uses large overestimations to compute the range of a result; according to its author, "of course, range arithmetic itself is a form of interval arithmetic, but here the interval is only crudely represented". The IntLab library primarily implements efficiently interval algorithms using MatLab, and, besides, mainly provides a type for arbitrary precision computations but implements few related functionalities: in version 3, only the exponential function and the π constant are available. Arbitrary precision interval arithmetic was also mentioned as an easy-to-implement extension to Brent's multiple precision package MP as early as 1981 [5]; however, Brent himself advises to use another package than MP now: "MP is now obsolescent. Very few changes to the code or documentation have been made since 1981! [...] In general, we recommend the use of a more modern package, for example David Bayley's MPP package or MPFR" (cf. `http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub043.html`).

Since none of the aforementioned packages implements a complete and really reliable arbitrary precision interval arithmetic, this led us to implement our own library.

In the next section, the need for arbitrary precision interval arithmetic is justified from a theoretical point of view and also by considering the use of bisection in interval algorithms. Then the MPFI (*Multiple Precision Floating-point Interval arithmetic library*) is presented: firstly the requirements on the underlying floating-point arithmetic are explained and then the functionalities and implementation choices are developed. Some benchmarks on Gaussian elimination performed on M-matrices allow to measure the overhead factor due to the handling of multiple precision numbers, which is 5 compared to the fixed-precision interval library `fi_lib`; the effects of increasing the precision are also shown. Lastly, some applications using MPFI are briefly presented in order to illustrate the various needs and benefits taken from this kind of arithmetic: guaranteed solutions of a problem solvable neither by exact nor by usual numerical methods, acceleration of an exact method and handling of inaccurate data, solution of a problem with arbitrary accuracy.

Theoretical background

The theoretical result underlying this idea can be found in [29]: let us denote by X an interval, by f a function and by F an interval extension of f, where F is given by a Lipschitz expression, let ε correspond to the current computing precision p: $\varepsilon = 2^{-p}$, then $F(X)$ overestimates $f(X)$ and the overestimation is bounded by

$$q(f(X), F(X)) \leq c_1 w(X) + c_2 \varepsilon \tag{1}$$

where q is the Hausdorff distance, $w(X)$ is the width of X and the constants c_1 and c_2 depend on F. This means that the computing precision can become a limiting factor and that being able to increase it can be an issue.

Furthermore, a classical procedure in interval analysis is the bisection one: if the output width is too large, then the inputs are split in two (or more) parts and the computation is repeated on each part. Bisection is often the last resort to get more accuracy, by reducing in formula (1) the quantity $w(X)$. In cases where $w(X) = u$ (which happened in our experiments on global optimization), only an increase in the computing accuracy, by "adding new floating-point numbers" between the endpoints of X, would have yielded a solution. Bisection can mainly be seen as a way to escape the wrapping effect by providing a paving of the sought set: the paving by a union of small boxes can be as accurate as desired and circumvents the overestimation induced by the inclusion into only one box. Bisection also allows to reduce the dependency problem, even if, in that respect, nothing supersedes the use of a good formulation.

It can be noticed that the rule of thumb "*to get more digits, one has to increase the computing precision by roughly the same number of digits*" can fail, for instance when computing a square root or more generally a $1/n$-th power close to 0. However, the rule of thumb becomes in such cases "*to get α more digits, one has to increase the computing precision by roughly $n\alpha$ digits*", as long as the computing precision enables to gain digits or with (ideal) infinite precision. In other words, in most cases an increase in the computing precision yields an improved accuracy on the results.

This is also the starting point of Müller's work on an effective simulation of a Real RAM [28], following the theoretical results by Brattka and Hertling [4] on the feasibility of a Real RAM. In Müller's work, a computation is performed and, if the final accuracy is not sufficient, then the whole computation is restarted with an increased precision; this is reiterated until the outputs are accurate enough.

The MPFI library

In order to implement an arbitrary precision interval arithmetic, a multiple precision floating point library was needed. By multiple precision, it is meant that the computing precision is not limited to the single or double precision of machine floating-point numbers; on the contrary, arbitrary precisions should be available. Furthermore, this computing precision must be dynamically adjustable to fulfill the accuracy needs. Another requirement for interval arithmetic is the outward rounding facility: this ensures that for each operation, the interval computed using floating-point arithmetic contains the interval obtained if exact real arithmetic were used. Even more desirable is *exact* directed rounding to avoid losing accuracy, *i.e.* the interval computed using floating-point arithmetic is the smallest one (for inclusion); however, it is rarely fulfilled for elementary functions. To sum up, compliance with the IEEE 754 standard for floating-point arithmetic, extended to elementary functions, is welcome.

The Arithmos project of the CANT team, U. Antwerpen, Belgium [6], or the MPFR library (*Multiple Precision Floating-point Reliable arithmetic library*), developed by the Spaces team, INRIA Lorraine, France [11], are such libraries. For portability and efficiency reasons (MPFR is based on GMP and efficiency is a motto for its developers) and also because of the availability of the source code, we chose MPFR. The corresponding library, named MPFI for *Multiple Precision Floating-point Interval arithmetic library* [31], is a portable library written in C for arbitrary precision interval arithmetic (source code and documentation can be freely downloaded). It is based on the GNU MP library and on the MPFR library and is part of the latter. The largest achievable computing precision is provided by MPFR and depends in practice on the computer memory. The only theoretical limitation is that the exponent must fit in an integer. Let us just say that it is possible to compute with numbers of several millions of binary digits if needed.

Intervals are implemented using their endpoints, which are MPFR floating-point numbers: this is not visible for the user but ensures that the swelling of intervals' widths is less important than with the midpoint-radius representation such as implemented by Rump in IntLab [33, 34]. Indeed, switching the rounding modes incurs no penalty with multiple precision arithmetic and the motivation for this choice in IntLab does not hold for MPFI: every multiple precision operation is a software one.

The specifications used for the implementation are based on the $IEEE$ standard [15]:

- an interval is a connected closed subset of $\mathbb{R}$;
- if *op* is an n-ary operation and $X_1, \ldots, X_n$ are intervals, the result of

$op(X_1, \ldots, X_n)$ is an interval such that: $\{op(x_1, \ldots, x_n), x_i \in X_i\} \subset op(X_1, \ldots, X_n)$;

- in case $op(x_1, \ldots, x_n)$ is not defined, then a `NaN` ("Not a Number", which stands for an invalid operation) is generated, *i.e.* the intersection with the domain of *op* is not taken prior to the operation;
- each endpoint carries its own precision (set at initialization or modified during the computations).

The arithmetic operations are implemented and all functions provided by MPFR are included as well (trigonometric and hyperbolic trigonometric functions and their reciprocals). Conversions to and from usual and GMP data types are available as well as rudimentary input/ouput functions. The code is written according to GMP standards (functions and arguments names, memory management).

The planned functionalities, that will be added in a near future, include a C++ interface à la Profil/BIAS [23] for ease of use, basic tools for linear algebra (vector and matrix data types, additions and multiplications) and automatic differentiation (forward differentiation by overloading operators and functions).

Benchmarks

According to our experiments, the MPFI implementation is as efficient as theoretically expected: for the Gaussian elimination performed on a "random" M-matrix of dimension 300 and followed by a computation of the residual to check the result, the overhead factor is close to 2, compared to MPFR.

To give a precise idea of the additional cost caused by the use of a multi-precision arithmetic, we compare MPFI with `fi_lib`. The library `fi_lib` [25] is based on the native *double* type for floating-point numbers; its main features are on the one hand to avoid switching rounding modes, by adding or subtracting one ulp to each endpoint after each operation (however, a faithful rounding is assumed), and on the other hand to implement software evaluations of elementary functions that return enclosing intervals and to avoid the use of elementary functions provided by the computer or the compiler, which return results with unknown and possibly large errors. To take into account the point of view and the constraints of a user, we implemented a simple Gaussian elimination method to solve linear systems, using both packages. The functions and the data structures implemented in `fi_lib` and MPFI have been homogenized (embedded into classes having exactly the same specifications - we directly used only the C functions and data structures from `fi_lib`) so that the algorithms using these arithmetic libraries are strictly identical. In order to measure the quality of the implementation, we embedded in the same way hardware "double" coefficients (floating-point numbers, not intervals) as well and ran the resulting algorithm.

On the one hand, the computation time includes the cost of the various loops and function calls, but on the other hand it also includes the memory management aspects induced by the MPFI package (the numbers are always dynamically allocated).

The timings in seconds of our programs and ratios compared to `fi_lib` are the following on an Athlon 1GHz processor:

double	`fi_lib`	MPFI 53 bits	
0.42	4.39	21.75	
		4.95	
MPFI 63 bits	MPFI 127 bits	MPFI 255 bits	MPFI 511 bits
22.35	24.11	34.09	51.61
5.09	5.49	7.76	11.75
MPFI 1023 bits	MPFI 2047 bits	MPFI 4095 bits	
112.30	285.57	817.05	
25.58	65.05	186.11	

Comments on the timings
The first remark is that the overhead induced by MPFI in 53 bits mode is reasonable (a factor less than 5) compared to a package that uses hardware floating point arithmetic. This overhead is due to the dynamic management of memory (allocation of numbers) in GMP – MPFI is built upon MPFR which is based on GMP –, which is comparable to the cost of arithmetic operations for small precisions. However, highly optimized functions written in assembly language allow to perform efficiently operations with data encoded with few machine words and this prevents this overhead from being larger for small precisions. This explains why the overhead implied by a doubling of the precision is noticeable starting from 1023 bits (32 machine words) only.

Let us also highlight the overhead incurred by successive doublings of the precision: the overhead factor is less or equal to 3, which is due to the cost of Karatsuba multiplication in multiple precision, and the factor between MPFR and MPFI remains constant and equal to 2 for this program on M-matrices (it is upper bounded by 4 when general multiplications occur).

Comments on the accuracy of the results
In 53 bits mode, the intervals produced by MPFI are always strictly included into those provided by the `fi_lib` package, whether the solution or the residual is concerned. This is due to the use, in `fi_lib`, of a software "blowing" of the intervals to avoid frequent switchings of rounding modes. The results obtained by `fi_lib` are extremely close to those computed by MPFI using 52 bits.

Being able to work with 511 bits of precision (*i.e.* 10 times more than with hardware floating-point numbers) loosing only a factor about 12 compared to a more conventional interval arithmetic library based on hardware doubles pleads in favour of MPFI: it is comfortable to develop an algorithm and to correct a lack of accuracy by increasing the precision of the underlying floating-point arithmetic; this lack of accuracy can be attributed either to a quick and not very careful expression of the problem (in the first stage of the development for instance) or to theoretical reasons such as those mentioned in the second section of this paper.

Applications

There are several motives to use interval arithmetic in scientific calculation. One can think for example of the following points: validation of numerical algorithms (study of the precision of the output when it is not known from the theory), exact calculations (where the qualitative aspect of the result may be

more important than the quantitative aspect), algorithms requiring global information and thus designed to work with interval arithmetic. We show in the few following examples that the use of multi-precision floating-point is crucial in many situations.

As mentioned in the introduction, having a precise interval arithmetic may be useful for studying ... other arithmetics, and namely the accuracy of the result. One good example is perhaps the solution of the direct kinematics problem for parallel robots [26]. This problem can be expressed as an algebraic system depending on 7 to 12 variables (several formulations exist) and no known numerical method allows, nowadays, to solve it in the general case. We analyzed some computational strategies which are usually employed (mainly eigenvalue computations) but that fail on some examples, by simply replacing the hardware floating-point arithmetic by MPFI. The result was surprising: using classical arithmetic, none of the solutions could be approximated, whereas all the solutions were computed with a good accuracy using 128 bits of precision with MPFI. Furthermore, only few exact methods can solve efficiently this problem at present. For the experiments described above, we had to construct some matrices using exact computations (Gröbner bases) to avoid an accumulation of numerical errors. This is one way to mix exact and numerical computations.

Other experiments have put in evidence that interval arithmetic may replace, after some additional theoretical work, multi-precision rational numbers in exact computations. Let us illustrate this with methods based on Descartes's rule of signs.

Basically (see [32] for example), the exact version of the so-called Uspensky's algorithm is already known to be very efficient for isolating the real roots of univariate polynomials: it can deal with orthogonal polynomials like Wilkinson's or Chebyshev's ones, etc. of degree greater than 2000. To summarize, the algorithm is based on Descartes's rule of signs which is defined below.

Notation 1 *We denote by* $\mathrm{sign}(a)$*, the* sign *of an element* $a \in \mathbb{R}$*, as being* 0 *if* $a = 0$*,* 1 *if* $a > 0$ *and* -1 *if* $a < 0$*, and define the* number of sign changes $V(a)$ *in the list* $a = (a_1, \cdots, a_k)$ *of elements of* $\mathbb{R} \setminus \{0\}$ *by induction over* k*:*

$$V(a_1) = 0, V(a_1, \cdots, a_k) = \begin{cases} V(a_1, \cdots, a_{k-1}) + 1 & \text{if } \mathrm{sign}(a_{k-1}a_k) = -1, \\ V(a_1, \cdots, a_{k-1}) & \text{otherwise.} \end{cases}$$

We extend this notation to a list of elements in $\mathbb{R}$ *that may contain zeroes: if* b *is the list obtained by removing zeroes in* a*, we define* $V(a) = V(b)$*.*

Using the above notations, Descartes's rule of signs is the following:

Theorem 1 (Descartes's rule of signs) *Let* $P = \sum_{i=0}^{d} a_i x^i$ *be a polynomial in* $\mathbb{R}[x]$*. If we denote by* $V(P)$ *the number of sign changes in the list* $(a_0, \ldots, a_d)$ *and* $\mathrm{pos}(P)$ *the number of positive real roots of* P *counted with multiplicities, then* $\mathrm{pos}(P) \leq V(P)$*, and* $V(P) - \mathrm{pos}(P)$ *is even.*

Let us assume that P is a square-free polynomial of degree d in $\mathbb{R}[x]$ with all its roots in $]0, 1[$. Defining $P_{k,c} = 2^{kd} P(\frac{x+c}{2^k})$, if Descartes's rule of signs gives 1 (resp. 0) when applied to the polynomial $(x+1)^d P_{k,c}(1/(1+x))$,

then the polynomial P has exactly one root (resp. no root) in the interval $]\frac{c}{2^k}, \frac{c+1}{2^k}[$. Theorems due to Collins/Akritas [9] and Vincent [36] state that for sufficiently large values of k, Descartes's rule of signs when applied to the polynomial $(x+1)^d P_{k,c}(1/(1+x))$ will always give 0 or 1.

The main remark is that we are only interested in the signs of coefficients, and thus the accuracy needed for the result of computations can be poor. However, the transformations $x \rightarrow x/2$ used during the computations forbid the use of hardware floats (even interval arithmetic based on hardware floats) for solving polynomials of high degree, because of limitations on the exponents' range. Moreover, the algorithm will terminate only if the polynomial P is square-free: this condition has no precise meaning when the coefficients of P are represented by floats or intervals.

Moreover, when replacing rational coefficients with intervals, two problems may occur:

- a lack of numerical accuracy may induce problems in sign determinations (0 can appear in intervals);
- the polynomial may not be square-free (say a polynomial with interval coefficients represents a set of polynomials that may contain a non square-free one).

As shown in [32], these two situations lead to the same problem: some sign determinations will not be possible, which means that, if a sign determination failure is used as a stopping criterion, then the algorithm will always terminate. The final result will be a set of isolating intervals and a set of intervals for which no decision is possible (they may or may not contain real roots). One can then increase the precision of the arithmetic to continue the computations: one important feature is that the computation can be restarted exactly where it failed, *i.e.* there is no need to restart the computation from the beginning. If the coefficients of the initial polynomial are known exactly, such a process will always give a complete and exact result.

This algorithm is a very good example of symbolic/numeric computations. The multi-precision arithmetic speeds up the computations in general but also allows to deal with polynomials whose coefficients are not exactely known (approximate coefficients, real algebraic numbers, etc.).

Revol [30] has implemented interval Newton algorithm [14] and has adapted it to multiple precision computations. In this example, the main advantage of using MPFI is that one is no more limited by the computing precision: indeed, one can impose arbitrary accuracy on both the root's approximation and the residual in Newton's algorithm [19]. Let us insist on the fact that the two aforementioned implementations managed to adapt dynamically the precision to the computing needs without restarting the whole program. This desirable feature will be sought after for future implementations of other algorithms.

Conclusion

MPFI is a library for multiple precision interval arithmetic which fully satisfies the containment requirement and allows to compute with arbitrary precision.

It exhibits acceptable overheads compared to fixed-precision interval libraries and its performances do not drop dramatically when the computing precision increases. MPFI is written in C and built upon MPFR and GMP and can be freely downloaded. It is still under development: new facilities such as automatic differentiation and linear algebra will be added in the near future.

It has enabled us to implement and test some algorithms, such as determining all solutions of the general case of the direct kinematic problem for parallel robots, isolating real roots of polynomials of large degree and with possibly inaccurately known coefficient or approximating with arbitrary accuracy zeroes of a function using an adapted interval Newton algorithm. The computing precision has been dynamically adapted in order to fulfill the computational needs, without requiring to restart the program from the very beginning. This will be pursued with a careful study of the solution of linear systems and of global optimization of continuous functions [13, 18]. Applications such as parameter estimation in automatics [16] will offer the opportunity to gain further insight in the development of new algorithms.

References

[1] O. Aberth and M.J. Schaefer. Precise computation using range arithmetic, via C++. *ACM TOMS*, 18(4):481–491, December 1992.

[2] G. Alefeld and J. Herzberger. *Introduction to interval analysis*. Academic Press, 1983.

[3] M. Berz and J. Hoefkens. Verified high-order inversion of functional dependencies and interval Newton methods. *Reliable Computing*, 7:1–20, 2001.

[4] V. Brattka and P. Hertling. Feasible real random access machines. *J. of Complexity*, 14(4):490–526, 1998.

[5] R. P. Brent. A Fortran multiple-precision arithmetic package. *ACM TOMS*, 4:57–70, March 1978.

[6] CANT Research Group. Arithmos: a reliable integrated computational environment. University of Antwerpen, Belgium, `http://win-www.uia.ac.be/u/cant/arithmos`, 2001.

[7] M. Ceberio and L. Granvilliers. Solving Nonlinear Systems by Constraint Inversion and Interval Arithmetic. In *Int. Conf. on Artificial Intelligence and Symbolic Computation (AISC'2000), LNAI 1930*.

[8] J.-M. Chesneaux, S. Guilain, and J. Vignes. *La bibliothèque CADNA : présentation et utilisation*. `http://www-anp.lip6.fr/cadna`, 1996.

[9] G. Collins, and A. Akritas. Polynomial real root isolation using Descartes' rule of signs. In *SYMSAC* (1976), pp. 272–275.

[10] A.E. Connell and R.M. Corless. An experimental interval arithmetic package in Maple. In *Num. Analysis with Automatic Result Verification*, 1993.

[11] D. Daney, G. Hanrot, V. Lefèvre, F. Rouillier, and P. Zimmermann. The MPFR library. `http://www.mpfr.org`, 2001.

[12] I. Geulig and W. Krämer. Intervallrechnung in Maple - Die Erweiterung intpakX zum Paket intpak der Share-Library. Technical Report 99/2, Universität Karlsruhe, 1999.

[13] E. Hansen. *Global optimization using interval analysis.* Marcel Dekker, 1992.

[14] E. Hansen and R.I. Greenberg. An interval Newton method. *J. of Applied Math. and Computing*, 12:89–98, 1983.

[15] T.-J. Hickey and Q. Ju and M.-H. Van Emden Interval Arithmetic: from Principles to Implementation *J. of ACM*, 2002.

[16] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied interval analysis.* Springer Verlag, 2001.

[17] R.B. Kearfott, M. Dawande, K.-S. Du and C.-Y. Hu. Algorithm 737: INTLIB: a portable Fortran 77 interval standard-function library. *ACM TOMS*, 20(4):447–459, 1994.

[18] R.B. Kearfott. *Rigorous global search: continuous problems.* Kluwer, 1996.

[19] R.B. Kearfott and G.W. Walster. On stopping criteria in verified nonlinear systems or optimization algorithms. *ACM TOMS*, 26(3):373–389, September 2000.

[20] J. Keiper. Interval arithmetic in Mathematica. *Interval Computations*, (3), 1993.

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

[22] W. Krämer, U. Kulisch and R. Lohner. *Numerical toolbox for verified computing II – Advanced Numerical Problems.* to appear (1998).

[23] O. Knueppel. PROFIL/BIAS - a fast interval library. *Computing*, 53(3-4):277–287, 1994.

[24] P. Langlois. Automatic linear correction of rounding errors. *BIT Numerical Algorithms*, 41(3):515–539, 2001.

[25] M. Lerch, G. Tischler, J. Wolff von Gudenberg, W. Hofschuster and W. Krämer. *The interval library filib++ 2.0. Design, features and sample programs.* Preprint 2001/4, `http://www.math.uni-wuppertal.fr/org/WRST/software/filib.html`, Universität Wuppertal, Germany 2001.

[26] J.P. Merlet. *Les robots parallèles.* Hermès Paris, 1990, Robotique.

[27] R.E. Moore. *Interval analysis.* Prentice Hall, 1966.

[28] N. Müller. The iRRAM: Exact Arithmetic in C++. In *Workshop on Constructivity and Complexity in Analysis, Swansea, 2000.*

[29] A. Neumaier. *Interval methods for systems of equations.* Cambridge University Press, 1990.

[30] N. Revol. Newton's algorithm using multiple precision interval arithmetic. Research report 4334, INRIA, (submitted to Numerical Algorithms) 2001.

[31] N. Revol and F. Rouillier. The MPFI library. `http://www.ens-lyon.fr/~nrevol`, 2001.

[32] F. Rouillier and P. Zimmermann. Efficient isolation of polynomial real roots. to appear in *J. of Computational and Applied Math.*, 2002.

[33] S. Rump. *Developments in Reliable Computing, T. Csendes ed.*, chapter INTLAB - Interval Laboratory, pages 77–104. Kluwer, 1999.

[34] S. Rump. Fast and parallel interval arithmetic. *BIT*, 39(3):534–554, 1999.

[35] Sun Microsystems, Inc. *C++ Interval Arithmetic Programming Reference*. 2000.

[36] A.-H. Vincent. Sur la résolution des équations numériques. *Journal de Mathématiques Pures et Appliquées* (1836), 341–372.