Motivations for an arbitrary precision interval arithmetic and the MPFI library

N. Revol (nathalie.revol@inria.fr)[*]
INRIA, Project Arenaire, LIP (CNRS/ENSL/INRIA/UCBL), École Normale Supérieure de Lyon, 69364 Lyon Cedex 07, France

F. Rouillier (fabrice.rouillier@inria.fr)[†]
Salsa Project, INRIA Rocquencourt and LIP6, France

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 acceptable. 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, accuracy

AMS Mathematics Subject Classification: 65G30

1. Motivations and state of the art

Computing with interval arithmetic [3, 25, 31] gives guarantees on a numerical result. 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 $[3.14159, 3.14160]$. Measurement errors also can be taken into account. The advantages of interval arithmetic are numerous. Computed results are validated, and rounding errors are taken into account, since computer implementations perform outward rounding. Last but not least, interval arithmetic provides global information: for instance, it provides the range of a function over a whole set S, which is a crucial information for global optimization. Such properties cannot be reached without set computing: interval arithmetic computes with sets and is easily available.

[*] This work was done while N. Revol was a member of the ANO Laboratory, University of Lille, France, on sabbatical leave within the Arenaire project.
[†] This work was done while F. Rouillier belonged to the Spaces project, LORIA and LIP6, France.

However, in spite of the improvements in interval analysis, the problem of overestimation, *i.e.* of enclosures which are far too large and thus of little practical use, seems to plague interval computations. With fixed-precision floating-point arithmetic, results can be wide even when the input data are provided with machine precision; a remedy for this phenomenon consists in computing with a higher precision. This proposal is the core of the MPFI library (MPFI stands for *Multiple Precision Floating-point Interval*), a library implementing arbitrary precision interval arithmetic which is described in this paper.

This quest for extra accuracy can be found primarily in the development of algorithms for interval analysis: use of centered forms and slopes instead of direct interval evaluation, preconditioners, and many other directions [5]. Real-world applications where extra accuracy is required are to be found in control theory (we have indeed received requests for a tool that would 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 [27], Intlib [21]...) or compilers (Sun Forte, *e.g.* C++ [41]) are based on *double* floating-point numbers and do not propose arbitrary precision interval arithmetic. The XSC languages [25] include a long accumulator type, initially for accurate dot product. It has also been used for Horner evaluation of a polynomial in the interval Newton algorithm for example [26]. The maximal precision is thus limited, since the long accumulator type relies on the double type, whose range of exponents is $\{-1022, \cdots, 1023\}$. Several arbitrary precision interval packages are available, for instance intpak [11] and intpakX [15] for Maple or a similar package for Mathematica [24]. These packages cannot be considered as reliable: unverified assumptions on the roundings of elementary functions (0.6 ulp for intpak in Maple, 1 ulp for Mathematica), incorrect roundings[1] and several other undue assumptions can be found. Other works include “range arithmetic” [1, 2] and IntLab [40]. The *range* library is a (now quite old) arbitrary precision library which primarily aims at indicating the number of correct digits rather than at performing interval arithmetic. However, it implements variable precision interval arithmetic for this purpose. It is a pioneering work, however the C++ used to implement it can no more be compiled by recent

[1] For instance, with 3 decimal digits, the rounding towards 0 of $1 - 9.10^{-5}$ gives 1 instead of 0.999 in Maple v6 to v8.

compilers. The IntLab library primarily implements efficient interval algorithms using MatLab, and additionally 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 1978 [8]. It was implemented in 1980 [43], using the Augment preprocessor (which replaced arithmetic operators by calls to the appropriate functions, as overloading was not available). However, the precision was not really dynamic, since the memory storage used for every number corresponded to the maximal precision indicated at compile time. Furthermore, Brent himself now advises to use another package than MP: "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 section 2, we recall the theorem which justifies our approach from a theoretical point of view. It states that using arbitrary precision yields arbitrary accuracy. The MPFI (*Multiple Precision Floating-point Interval arithmetic*) library is then presented in section 3. The requirements on the underlying floating-point arithmetic are explained before the functionalities and implementation choices are developed. Then, in section 4, some benchmarks on Gaussian elimination performed on M-matrices allow measuring the overhead factor due to the handling of arbitrary precision numbers. This factor is 5 compared to the fixed-precision interval library `fi_lib`; the effects of increasing the precision are also shown. Lastly, some applications using our MPFI library are presented in section 5, in order to illustrate the power of this kind of arithmetic. One could obtain guaranteed solutions to a problem not solvable by usual numerical methods, acceleration of an exact method and handling of inaccurate data, and finally solution to a problem with arbitrary accuracy.

2. Influence of the computing precision

The idea underlying MPFI is expressed by the following theorem [33].

THEOREM 1. *Let $\boldsymbol{x}$ be an interval, f a function and $\boldsymbol{f}$ an interval extension of f, let us assume that $\boldsymbol{f}$ is given by a Lipschitz expression. Let ε correspond to the current computing precision p: $\varepsilon = 2^{-p}$. Then $\boldsymbol{f}(\boldsymbol{x})$ overestimates $f(\boldsymbol{x})$, and the overestimation is bounded by*

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

where q is the Hausdorff distance, $w(\boldsymbol{x})$ is the width of $\boldsymbol{x}$, and the constants c_1 and c_2 depend on $\boldsymbol{f}$.

This means that the computing precision p can become a limiting factor to the accuracy of the result for small intervals and that being able to increase the computing precision can become an issue.

A first problem where arbitrary precision proves useful is the global optimization of a continuous function. Hansen's algorithm [17] highlights the fact that bisection is sometimes the last resort to get improvements. Of course, pruning techniques are extremely useful and should not be neglected. 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 one single box.

A second example comes from linear algebra: Hansen and Sengupta algorithm [16] "solves" a linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ with $\boldsymbol{A}$ an interval matrix and $\boldsymbol{b}$ an interval vector, *i.e.* it computes an enclosure of the convex hull of $\{x : \exists A \in \boldsymbol{A}, \exists b \in \boldsymbol{b}, Ax = b\}$. This algorithm firstly computes C, an approximation of $(\mathrm{mid}\boldsymbol{A})^{-1}$, and then applies Gauss-Seidel iterations to $C\boldsymbol{A}\boldsymbol{x} = C\boldsymbol{b}$ until convergence is reached. This algorithm is based on the idea (or hope) that, since $C\boldsymbol{A}$ contains the identity matrix, it is diagonally dominant and thus Gauss-Seidel iteration is a contraction. This means that an arbitrary accuracy can be reached... if arbitrary computing precision is available.

The theorem above is also the starting point of Müller's work on an effective simulation of a Real RAM [32], following the theoretical results by Brattka and Hertling [7] on the feasibility of a Real RAM. In Müller's work, a computation is performed. If the final accuracy is not sufficient, then the whole computation is restarted with a higher computing precision; this process is repeated until the outputs are accurate enough.

3. 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 can be set arbitrarily and is not limited to the single or double precision of machine floating-point numbers. Furthermore, this computing precision must be dynamically adjustable in response to the accuracy needed. Another requirement for interval arithmetic is the outward rounding facility, ensuring that for each operation, the interval computed using floating-point arithmetic contains the interval obtained should exact real arithmetic be used. Very desirable, *exact* directed rounding avoids losses in accuracy, *i.e.* the interval computed using floating-point arithmetic is the smallest one for inclusion. However, it is rarely available for elementary functions. To sum up, compliance with the IEEE 754 standard for floating-point arithmetic, extended to elementary functions, is welcome.

This can be found in the Arithmos project of the CANT team, U. Antwerpen, Belgium [9], or in the MPFR library (*Multiple Precision Floating-point Reliable arithmetic library*), developed by the Spaces team, INRIA Lorraine, France [13]. 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 interval library, named MPFI for *Multiple Precision Floating-point Interval arithmetic library* [35], is a portable library written in C for arbitrary precision interval arithmetic (source code and documentation can be freely downloaded at the url `http://perso.ens-lyon.fr/nathalie.revol/software.html`). It is based on the GNU MP library and on the MPFR library. The largest achievable computing precision is determined by MPFR and depends in practice on the computer memory. The only theoretical limitation is that the exponent must fit in a machine integer. It suffices to 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 [39, 40]. Every interval arithmetic operation using the endpoint representation switches the rounding mode. With the midpoint-radius representation, matrix operations can be performed with only one change of rounding mode per matrix operation. This means that IntLab fully exploits the processor's pipeline, whereas libraries based on the representation by endpoints do not. As switching the rounding modes incurs no computing time penalty with arbitrary precision arithmetic, the motivation for this choice in IntLab does not hold for MPFI, since in MPFI, every multiple precision operation is a software one.

The specifications used for the implementation are based on the IEEE 754 standard [19]:

- an interval is a connected closed subset of $\mathbb{R}$;
- if *op* is an n-ary operation and $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ are intervals, the result of $op(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$, the operation *op* performed with interval arguments, is an interval such that: $\{op(x_1, \ldots, x_n), x_i \in \boldsymbol{x}_i\} \subset op(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$;
- 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/output functions. The code is written according to GMP standards (functions and arguments names, memory management).

The planned functionalities, that are under development, include a C++ interface à la PROFIL/BIAS [27] 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).

4. Benchmarks

According to our experiments, the MPFI implementation is as efficient as theoretically expected. For 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, whatever the precision.

To give a precise idea of the additional cost caused by the use of an arbitrary precision arithmetic, we compare MPFI with `fi_lib`. The library `fi_lib` [29] is based on the native *double* type for floating-point numbers. A first important feature of `fi_lib` is the avoidance of switching rounding modes, by adding or subtracting one ulp to each endpoint after each operation (however, a faithful rounding is assumed). Another important point is that `fi_lib` avoids the use of elementary functions provided by the computer or the compiler, which return results with unknown and possibly large errors. Indeed, `fi_lib` implements software evaluations of elementary functions that return enclosing intervals. 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 (which constitutes a higher percentage of the computing time of the `fi_lib` computation than of the MPFI one), but on the other hand it also includes the memory management aspects induced by the MPFI package (as the numbers are always dynamically allocated, this constitutes a non negligible part of MPFI computing time).

The timings in seconds of our programs and ratios compared to `fi_lib` on an Athlon 1GHz processor (using gcc v3.04 with options -O2, `fi_lib` 1.1, GMP 4.1, MPFR 2.0.1 and MPFI 1.0) are given in Table I.

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, which is comparable to the cost of arithmetic operations for small precisions.

A second remark is that this overhead increases slowly as the computing precision increases, up to 1024 bits (32 machine words). Indeed, highly optimized functions written in assembly language perform efficiently operations with data encoded with few machine words; this prevents the overhead from being too large for small precisions.

Table I. Timings using floating-point double precision, `fi_lib` interval arithmetic and MPFI arbitrary precision interval arithmetic on Gaussian elimination on a M-matrix of dimension 300.

Arithmetic	double	`fi_lib`	MPFI	MPFI	MPFI
Prec. (in bits)	53	53	53	63	127
Time (in s)	0.42	4.39	21.75	22.35	24.11
Ratio	0.10	1	4.95	5.09	5.49
Arithmetic	MPFI	MPFI	MPFI	MPFI	MPFI
Prec. (in bits)	255	511	1023	2047	4095
Time (in s)	34.09	51.61	112.30	285.57	817.05
Ratio	7.76	11.75	25.58	65.05	186.11

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. Furthermore, 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 bit mode, the intervals produced by MPFI are always strictly included in 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 of 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 section 2.

5. Applications

There are several motivations to use interval arithmetic in scientific calculation. One can think for example of the following aspects: validation of numerical algorithms (study of the precision of the output when it is not known from the theory or, in the example presented below, remedy to numerical failures), 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 interval arithmetic is crucial.

5.1. Solving algebraic systems - parallel robots

One of the motivations for the development of MPFI was to propose effective methods for solving algebraic systems with a finite number of solutions.

We focus on strategies relying on Gröbner bases calculations (see [12] for an introduction), which can handle realistic problems in numerous application fields. A Gröbner basis characterizes the ideal associated to a system of equations by a basis of the ideal and a function of reduction modulo this ideal. Once a Gröbner basis is computed, various approaches yield numerical approximations of the real or complex solutions of the system.

Let $S = \{f_1 = 0, \dots f_s = 0\}$ be an algebraic system with rational coefficients depending on n variables $X_1, \dots, X_n$, let I be the ideal generated by $\{f_1, \dots, f_s\}$ in $\mathbb{Q}[X_1, \dots, X_n]$ and G any Gröbner basis of I. One can *read* on G, without any further computation, whether S has a finite number of complex solutions. In the case of a finite number of solutions, the algebra $A = \mathbb{Q}[X_1, \dots, X_n]/I$ is a finite dimensional $\mathbb{Q}$-vector space, and the monomials of $\mathbb{Q}[X_1, \dots, X_n]$ which are irreducible modulo G form a basis of A.

A first approach to solve the algebraic system S reduces to a computation of eigenvalues. Indeed, given any polynomial $q \in \mathbb{Q}[X_1, \dots, X_n]$, the multiplication by q in A is a linear application in a vector space of finite dimension, and the corresponding matrix M_q can be explicitly constructed. The eigenvalues of M_q are exactly the scalars $q(\alpha_1), \dots, q(\alpha_d)$, where $\alpha_1, \dots, \alpha_d$ are the complex roots of S. As a consequence, the simultaneous computation of all eigenvalues of $M_{X_1}, \dots, M_{X_n}$ provides an approximation of all the complex solutions of S, as detailed in [4]

Another approach continues with exact computations and computes the so-called Rational Univariate Representation (RUR) as suggested in [36]. A RUR has the following shape:

$$\begin{cases} h(T) &= 0, \\ X_1 &= h_1(T)/h_0(T), \\ & \vdots \\ X_n &= h_n(T)/h_0(T), \end{cases}$$

where T denotes a *new* variable, h and $h_i,\ i = 0 \ldots n$, belong to $\mathbb{Q}[T]$, and h and h_0 are coprime.

In order to compute a certified approximation of the solutions, one has first to compute the solutions of the equation $h(T) = 0$. If we are interested in computing the real solutions of the system, the real solutions of $h = 0$ can be described by isolating intervals as explained in section 5.2. If these intervals can be sufficiently refined so that h_0 never vanishes on them (which is theoretically possible since h and h_0 are coprime), then it suffices to evaluate the rational functions h_i/h_0 on these intervals to compute isolating boxes containing the real solutions of S.

As mentioned above, having a precise interval arithmetic may be useful for studying other arithmetics, and namely the accuracy of the result or simply for overcoming a numerical instability. An illustration of this comes from the solution of the direct kinematics problem for parallel robots [30].

In our problem, the studied parallel manipulators were hexapod manipulators: they consist in a mobile platform and a base, linked by 6 rigid kinematic chains, cf. figure 1. The joints between the kinematic chains and the platforms are assumed to be concentric frictionless ball joints. The goal is to determine all possible positions of the manipulator (up to 40); these positions must satisfy some constraints, such as the length of the kinematic chains, which are fixed, or the relative position of the joints on each platform.

Different modellings yield different formulations for this problem. A first class of modelling consists in considering two Cartesian coordinate systems, one for each platform: (O, x, y) for the basis, fixed platform and (O', x', y') for the upper, mobile platform. The position of the upper platform is determined by the vector OO' (3 unknowns) and by the rotation of the upper platform from a reference position (9 unknowns for the coefficients of the rotation matrix, or 4 coefficients if quaternions are used). The constraints are the fixed lengths of the kinematic chains (6 polynomial constraints of degree 2) and the fact that the coefficients of the rotation do indeed define a rotation (7 polynomial constraints of degree 3 in the 9 unknowns, or 1 constraint of degree 2 with the quater-

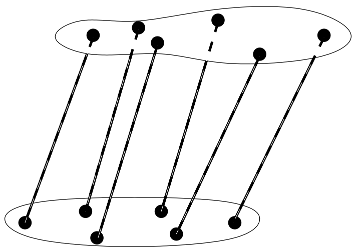

Figure 1. Parallel robot: the two platforms linked by rigid kinematic chains.

nion approach). A second class of modelling determines the position of the mobile platform from the position of 3 specific points C_1, C_2, C_3 on the mobile platform: this leads to a system of 9 equations (6 polynomial equations of degree 2 for the fixed lengths of the kinematic chains and 3 polynomial equations of degree 2 for the fixed distances between the 3 chosen points C_1, C_2, C_3) in 9 unknowns (the coordinates of C_1, C_2, C_3 in a fixed coordinate system).

In the case of the hexapod manipulator, the RUR-based computation, performed by the RS software [37], provides all solutions. More precisely, all real roots are enclosed with arbitrary accuracy using the algorithm described in section 5.2. The considered polynomial is the characteristic polynomial of the matrix, computed using computer algebra methods. Isolation of the real roots is performed with 64 bits of computing precision, and results have 32 correct bits. On the contrary, floating-point numerical computations of eigenvalues using NAG or Lapack libraries failed. They returned the correct number of roots, possibly complex, but with no correct digits. However, if the floating-point arithmetic used by these libraries is simply replaced by a more precise arithmetic, namely MPFI with 128 bits of mantissa, then these eigenvalues can be computed with a good accuracy.

This example illustrates on the one hand that symbolic and numeric computations can be successfully mixed. In this case the problem is symbolically pre-processed. On the other hand, classical numerical computations failed here, but precise interval ones did not. The extra precision allowed successful computation of the roots, and intervals guaranteed these solutions.

5.2. Isolation of polynomial real roots

Other experiments provide evidence that multiple precision interval arithmetic may replace, after some additional theoretical work, exact rational numbers in exact computations. Let us illustrate this with methods based on the Descartes rule of signs.

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 (see [38] for example). It can deal with orthogonal polynomials such as Wilkinson's or Chebyshev's, of degree greater than 2000. To summarize, the algorithm is based on Descartes rule of signs which is defined below.

NOTATION 1. *We 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*:*

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

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* $V(a) = V(b)$.

THEOREM 2 (Descartes 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 by* $\operatorname{pos}(P)$ *the number of positive real roots of* P *counted with multiplicities, then* $\operatorname{pos}(P) \leq V(P)$*, and* $V(P) - \operatorname{pos}(P)$ *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 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 [10] and Vincent [42] state that for sufficiently large values of k, Descartes rule of signs when applied to the polynomial $(x + 1)^d P_{k,c}(1/(1 + x))$ always yields 0 or 1.

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.

Furthermore, 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 [38], 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.

In [38], an algorithm based on the previous considerations is presented: this algorithm uses MPFI to evaluate the sign of the coefficients of the transformed polynomial $(x+1)^d P_{k,c}(1/(1+x))$; it doubles dynamically its computing precision when the determination of a sign fails.

Here we only present some results obtained on Laguerre and Mignotte polynomials, since they exhibit different features. These polynomials are known to be difficult, since they have a large number of roots or very close roots. More detailed comparisons with other algorithms, along with more experiments and more detailed results can be found in [38]. Tests were performed on a 1.5GB, 1GHz AMD Athlon processor.

Other efficient root finder programs are `MPSolve` *from Bini and Fiorentino* [4]*, and* `eigensolve` *from Fortune* [9, 8]*. Both programs give a floating-point approximation of* all *complex roots of a univariate polynomial. Since Fortune already compared* `eigensolve` *to* `MPSolve`*, we just compared our timings with* `eigensolve`*. For [. . .] Laguerre polynomials of degree 500, our program is 2-3 times faster than* `eigensolve v1.0`*. This speedup goes to more than 5 for the degree-500 Wilkinson polynomial. However,* `eigensolve` *is 100 times faster for Mignotte's polynomials: our algorithm does not use the fact that those polynomials are sparse.* (extract from [38], reference [4] corresponds to reference [6] in this paper and [9, 8] have been replaced by [14] here)

Compared to the same algorithm where all operations are performed using exact arithmetic, the results are as follows. For Laguerre polynomials of high degree (900 or 1000), the times are very close, whereas the algorithm using MPFI performs more than 15 times faster for Mignotte's polynomials. The algorithm using MPFI is only slightly

Table II. Number of roots isolated at a given precision, for Laguerre polynomials.

Degree	Computing precision (MPFI algorithm)					Exact algorithm	IS
	53	107	215	431	863		
200	0	0	2	198	0	0	3456
400	0	0	0	7	393	0	7648
600	0	0	0	0	70	530	12064
800	0	0	0	0	15	785	16864
900	0	0	0	0	3	897	18944
1000	0	0	0	0	0	1000	21056

more efficient than the exact one on Laguerre polynomials since most of the roots require a large working precision to be isolated. This need of a high precision will appear clearly on Table II. However, timings to the disadvantage of the algorithm using MPFI have not been observed.

Finally, Tables II and III illustrate the need for arbitrary precision. They display the number of roots which can be isolated only when a given precision is reached. The total number of roots is the sum of the numbers in a row. A maximal computing precision is set to 1024 (which may not be enough for the given examples). The algorithm doubles its computing precision when needed and switches to the exact algorithm when the maximal precision is reached. The last column, IS, gives the maximum bit-size of integers appearing in the exact algorithm.

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

Table III. Number of roots isolated at a given precision, for Mignotte polynomials.

Degree	Computing precision (MPFI algorithm)					Exact algorithm	IS
	53	107	215	431	863		
200	2	0	0	2	0	0	46400
400	2	0	0	0	2	0	185536
600	2	0	0	0	2	0	?

5.3. Approximation of the zeroes of a function

Revol [34] has implemented the interval Newton algorithm [18] and has adapted it to arbitrary precision computations. In this example, the algorithm is primarily intended for interval computations. It has been developed in order to enclose reliably every real zero of the function. Indeed, it uses an interval enclosure of the graph of the function and searches for zeroes in the intersection of this enclosure with the $Ox_1 \cdots x_n$ hyperplane, where $x_1, \cdots, x_n$ denote the variables.

With an interval arithmetic based on hardware floating-point numbers, the accuracy of the result is limited. In particular with a root of multiplicity $m > 1$ or a cluster of m zeroes, the accuracy on this zero is the computing precision divided by m. However, the interval Newton algorithm is based either on a contracting scheme or, if the contraction is not efficient enough, on a bisection. This implies that arbitrary accuracy can be reached, if only enough computing precision is available. This remark led us to implement the interval Newton algorithm in MPFI.

The interval Newton algorithm we derived is the following:

```
Input: $f$, $\boldsymbol{f'}$ an interval extension of the derivative of $f$, $X_0$ the initial search interval
Output: Res, a list of intervals that may contain the zeroes of $f$ in $X_0$.
Initialization: $\mathcal{L} = \{X_0\}$
Loop: while $\mathcal{L} \neq \emptyset$
        Suppress $(X, \mathcal{L})$
        one step of Newton on $X$:
            $(X_1, X_2) := (x - \frac{f(x)}{\boldsymbol{f'}(X)}) \bigcap X$ ($X_2$ can be empty)
        if the Newton step was not contracting enough then
            $(X_1, X_2) := bisect(X)$
        *increase the computing precision if needed* (*)
        *stopping test applied to $X_1$: insert $X_1$ in $\mathcal{L}$ or* Res (**)
        *ibid. for $X_2$*
```

In this algorithm one can recognize the classical interval Newton algorithm [18]. The modifications concern the increase of the computing precision (*) and the stopping criterion (**) (here, $\boldsymbol{f}$ denotes an interval extension of f):

– (*) the computing precision is increased when:

  - either the bisection is not possible: $X_1 = X$ because the endpoints of X are two consecutive floating-point numbers (for the current computing precision);
  - or the evaluation of $\boldsymbol{f}(X_1)$ is not narrower than $\boldsymbol{f}(X)$: $w(\boldsymbol{f}(X_1)) \geq w(\boldsymbol{f}(X))$ because the computing precision does not allow one to refine it (cf. theorem 1 of section 2).

– (**) stopping criterion: since arbitrary accuracy can be reached, the usual test "$w(X) \leq \varepsilon_X$ $\underline{\text{or}}$ $w(\boldsymbol{f}(X)) \leq \varepsilon_Y$" [23] is replaced by "$w(X) \leq \varepsilon_X$ $\underline{\text{and}}$ $w(\boldsymbol{f}(X)) \leq \varepsilon_Y$", where ε_X and ε_Y are thresholds given by the user.

– the termination of this new algorithm has been proven [34].

– if $\boldsymbol{f}$ contains interval coefficients, these coefficients should be handled as variables to be split, in order to enable refinements of the evaluations of $\boldsymbol{f}(X)$.

Some experiments have been performed on Chebyshev polynomials of degree up to 40, on the Wilkinson polynomial of degree 20 [34], and on a polynomial with a double root to illustrate the correct behaviour of this algorithm [28].

To conclude, 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.

6. Conclusion

MPFI is a library for arbitrary precision interval arithmetic which fully satisfies the containment requirement and allows one to compute with arbitrary precision. It exhibits acceptable overheads compared to fixed-precision interval libraries, and it is not extremely sensitive to increases of the computing precision. MPFI is written in C and built upon MPFR and GMP and can be freely downloaded. New facilities such as automatic differentiation and linear algebra are under development.

MPFI has enabled us to implement and test some algorithms. The first one, presented in section 5.1, determines all solutions of the general case of the direct kinematic problem for parallel robots. The second one, presented in section 5.2, isolates real roots of polynomials of large degree and with possibly inaccurately known coefficients. The third one, introduced in section 5.3, approximates with arbitrary accuracy zeroes of a function using an adapted interval Newton algorithm. In the two latter algorithms, the computing precision has been dynamically adapted in order to fulfill the computational needs, without requiring restarting the program from the very beginning. This work will be pursued with a careful study of the solution of linear systems and of global optimization of continuous functions [17, 22]. Applications such as parameter estimation in control theory [20] will offer the opportunity to gain further insight in the development of new algorithms.

Acknowledgements

We would like to thank George Corliss for the great job he did in checking this paper and offering numerous suggestions to improve it.

References

1. O. Aberth and M.J. Schaefer. Precise Computation using range Arithmetic, via C++. *ACM TOMS*, 18(4):481–491, 1992.
2. O. Aberth. *Precise Numerical Methods using C++*. Academic Press, 1998.
3. G. Alefeld and J. Herzberger. *Introduction to Interval Analysis*. Academic Press, 1983.
4. W. Auzinger and H.J. Stetter. An elimination algorithm for the computation of all zeros of a system of multivariate polynomial equations. *Int. Series in Numerical Mathematics,Birkhäuser*, 86:11–30, 1988.
5. M. Berz and J. Hoefkens. Verified High-Order Inversion of Functional Dependencies and Interval Newton Methods. *Reliable Computing*, 7:1–20, 2001.
6. D. Bini and G. Fiorentino. Numerical computation of polynomial tools: MPSolve - version 2.0. `http://www.dmi.unict.it/~marotta/Articoli/mpsolve.ps`, 1998.
7. V. Brattka and P. Hertling. Feasible real random access machines. *J. of Complexity*, 14(4):490–526, 1998.

8. R. P. Brent. A Fortran multiple-precision arithmetic package. *ACM TOMS*, 4:57–70, March 1978.
9. CANT Research Group. Arithmos: a reliable integrated computational environment. University of Antwerpen, Belgium, `http://win-www.uia.ac.be/u/cant/arithmos/`, 2001.
10. G. Collins and A. Akritas. Polynomial real root isolation using Descartes' rule of signs. In *SYMSAC* (1976), pp. 272–275.
11. A.E. Connell and R.M. Corless. An Experimental Interval Arithmetic Package in Maple. In *Num. Analysis with Automatic Result Verification*, 1993.
12. D. Cox, J. Little, and D. O'Shea. *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra.* Undergraduate texts in mathematics. Springer-Verlag, 1992.
13. D. Daney, G. Hanrot, V. Lefèvre, P. Pelissier, F. Rouillier, and P. Zimmermann. The MPFR library. `http://www.mpfr.org/`, 2001.
14. S. Fortune. Polynomial Root Finding Using Iterated Eigenvalue Computation. *Proc. ISSAC'01*, 121–128, 2001.
15. I. Geulig and W. Krämer. Intervallrechnung in Maple - Die Erweiterung intpakX zum Paket intpak der Share-Library. Technical Report 99/2, Universität Karlsruhe, 1999.
16. E. Hansen and S. Sengupta. Bounding solutions of systems of equations using interval analysis. *BIT*, 21:203–211, 1981.
17. E. Hansen. *Global Optimization Using Interval Analysis.* Marcel Dekker, 1992.
18. E. Hansen and R.I. Greenberg. An interval Newton method. *J. of Applied Math. and Computing*, 12:89–98, 1983.
19. T.-J. Hickey, Q. Ju and M.-H. Van Emden. Interval Arithmetic: from Principles to Implementation *J. of ACM*, 2002.
20. L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis.* Springer Verlag, 2001.
21. 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.
22. R.B. Kearfott. *Rigorous Global Search: Continuous Problems.* Kluwer, 1996.
23. R.B. Kearfott and G.W. Walster. On Stopping Criteria in Verified Nonlinear Systems or Optimization Algorithms. *ACM TOMS*, 26(3):373–389, 2000.
24. J. Keiper. Interval Arithmetic in Mathematica. *Interval Computations*, (3), 1993.
25. R. Klatte, U. Kulisch, C. Lawo, M. Rauch, and A. Wiethoff. *C-XSC: A C++ Class Library for Extended Scientific Computing.* Springer Verlag, 1993.
26. W. Krämer, U. Kulisch and R. Lohner. *Numerical Toolbox for Verified Computing II – Advanced Numerical Problems.* To appear (1998).
27. O. Knueppel. PROFIL/BIAS - A Fast Interval Library. *Computing*, 53(3-4):277–287, 1994.
28. P. Langlois and N. Revol. Validating polynomial numerical computations with complementary automatic methods. Submitted to *Mathematics and Computers in Simulation*, 4th revision. INRIA research report RR-4205, `http://www.inria.fr/rrrt/rr-4205.html`, May 2001.
29. M. Lerch, G. Tischler, J. Wolff von Gudenberg, W. Hofschuster and W. Krämer. *The Interval Library filib++ 2.0.* `http://www.math.uni-wuppertal.fr/org/WRST/software/filib.html`, Preprint 2001/4, Universität Wuppertal, Germany 2001.
30. J.P. Merlet. *Les Robots parallèles.* Hermès Paris, 1990, Robotique.

31. R.E. Moore. *Interval Analysis.* Prentice Hall, 1966.
32. N. Müller. The iRRAM: Exact Arithmetic in C++. In *Workshop on Constructivity and Complexity in Analysis, Swansea, 2000.*
33. A. Neumaier. *Interval Methods for Systems of Equations.* Cambridge University Press, 1990.
34. N. Revol. Interval Newton iteration in multiple precision for the univariate case. *Numerical Algorithms*, 34(2):417–426, 2003.
35. N. Revol and F. Rouillier. The MPFI library. `http://perso.ens-lyon.fr/nathalie.revol/software.html`, 2001.
36. F. Rouillier. Solving zero-dimensional systems through the rational univariate representation. *Journal of Applicable Algebra in Engineering, Communication and Computing*, 9(5):433–461, 1999.
37. F. Rouillier. RS / Real Solutions. `http://fgbrs.lip6.fr/~rouillie/Software/RS/`, 2000.
38. F. Rouillier and P. Zimmermann. Efficient isolation of polynomial real roots. *J. of Computational and Applied Math.*, 162(1):33–50, 2004.
39. S. Rump. Fast and parallel interval arithmetic. *BIT*, 39(3):534–554, 1999.
40. S. Rump. INTLAB - Interval Laboratory, in *Developments in reliable computing*, T. Csendes ed., pages 77–104. Kluwer, 1999.
41. Sun Microsystems. *C++ Interval Arithmetic Programming Reference.* 2000.
42. A.-H. Vincent. Sur la résolution des équations numériques. *Journal de Mathématiques Pures et Appliquées* (1836), 341–372.
43. J. M. Yohe. Portable Software for Interval Arithmetic. *Computing*, Suppl. 2, 211–229, 1980.