Fields Institute Communications
Volume **00**, 0000

Rigorous High Precision Interval Arithmetic in COSY INFINITY

Alexander Wittig
Department of Physics and Astronomy
Michigan State University
East Lansing, MI 48824

Martin Berz

Abstract. In this paper we present our implementation of a complete multiple machine precision interval package in FORTRAN 77. The operations are designed to perform optimally with up to about 100 significant digits. The implementation only requires IEEE 754 compliant floating point operations, and thus is highly portable to virtually any modern computer platform.

Intervals are stored as unevaluated floating point sums. Elementary operations such as addition and multiplication are based on exact floating point operations, while higher level operations such as division and square root are implemented based on those elementary operations. While there are several other non-rigorous high precision libraries based on these concepts, to the best of our knowledge this implementation is the first fully rigorous interval package.

Besides basic operations on high precision intervals, the library also provides a wide range of verified intrinsic functions such as sin, arctan, and ln. The library is integrated into the COSY INFINITY rigorous computation environment, providing fast and rigorous interval operations from within the COSY environment.

1 Introduction

Interval arithmetic has been a cornerstone of rigorous numerics for over 40 years[8]. Yet most readily available interval packages today are based on the floating point numbers provided by the underlying hardware. On current hardware platforms, intervals are thus limited to a relative width of about 10^{-15}[1]. For certain computations, more significant digits are needed for interval methods to succeed[13].

1991 *Mathematics Subject Classification.* Primary 65G30; Secondary 65G20, 65G50.
The first author was supported in part by DOE Grant #XXXXXX..

There already are some alternatives available, such as MPFI[9]. However, their use does not seem to be widespread in the interval community. Furthermore, the focus of MPFI is on very long numbers, with several thousand significant digits. Also most packages are written in C, and require several external libraries to be built and linked to FORTRAN code. This makes it harder to write portable and easily compilable code.

To overcome these obstacles, we have developed a high precision interval arithmetic that focuses on rigorous operations on numbers with about 100 significant digits. Our implementation is written entirely in FORTRAN 77, and is thus easily portable to all platforms supporting the IEEE754 standard on floating point operations.

Many other high precision libraries, rigorous or not, rely on large integer operations to emulate floating point numbers with large mantissa[5, 9]. We follow a different approach in our implementation. We store high precision numbers as an unevaluated sum of floating point numbers. Elementary operations on these expansions are reduced to operations on two floating point numbers. Those operations can then be performed fully rigorously within the floating point number framework using well known algorithms[4, 6]. While there are several other non-rigorous high precision packages available using these techniques (e.g. [10]), to the best of our knowledge none of them provide rigorous interval arithmetic.

Based on the elementary operations addition, subtraction and multiplication, we build the more complicated operations such as division and square root. Lastly, intrinsic functions such as sin, exp and arctan are computed based on their Taylor expansions.

2 Theory of High Precision Operations

We will begin by introducing the underlying elementary operations on floating point numbers, which we will use to build our high precision implementation on.

2.1 Floating Point Numbers.

To represent calculations on the real numbers on a computer, most modern processors use floating point numbers. We start with a precise definition of floating point numbers.

Definition 2.1 The set of all floating point numbers R is given by

$$R = \{m_z \cdot 2^{e_z} |\ 2^{t-1} \leqslant |m_z| < 2^t;\ \underline{M} < e_z < \overline{M}\},$$

where $t, \underline{M}$ and $\overline{M}$ are positive integer constants.

The constants $t, \underline{M}$ and $\overline{M}$ define the floating point number system. $\underline{M}$ and $\overline{M}$ limit the exponent range and thus the largest and smallest representable numbers. To make the following proofs easier to understand, we will assume that the exponent range is unlimited, i.e. $\underline{M} = -\infty$ and $\overline{M} = \infty$. This is, of course, not true for computer systems, where overflows and underflows of the exponent may happen. In our practical implementation we have to deal with those cases separately. The parameter t is the mantissa length in binary digits and thus defines the relative precision of the floating point system (see below).

In the following we will use floating point systems with different mantissa lengths which we will denote by R_t. Over- and underflows notwithstanding, we clearly have that $R_t \subset R_{t'}$ if $t \leqslant t'$. The lower bound requirement on the mantissa is called the normalization. With this additional requirement, the values represented by floating point numbers become unique. Mantissae with absolute value less than 2^{t-1} can be multiplied by a power of two so that they lie within the allowed range for the mantissa, while the exponent is adjusted accordingly.

Given any real number $r \in \mathbb{R}$ within the range of the floating point representation, we will denote by $\tilde{r} \in R$ the closest floating point number in the given system of floating point numbers. Then it follows readily from Definition 2.1 that

$$\frac{|r - \tilde{r}|}{|r|} < \epsilon_m = 2^{-t}.$$

The value ϵ_m is called the machine precision and is given by the length of the mantissa t.

Every floating point implementation has to provide at least the basic operations addition, subtraction, and multiplication. Clearly the mathematical result of any of those operations on two arbitrary floating point numbers $a, b \in R$ does not necessarily have to be in R. Thus, the floating point operations corresponding to $+, -, \times$ are not the same as their mathematical counterparts on the real numbers. Let $\oplus, \ominus, \otimes$ denote the floating point operations for $+, -, \times$.

Definition 2.2 Let $\odot$ denote one of the floating point operations $\oplus, \ominus, \otimes$ and $\bullet$ the same operation on the real numbers.
The operation $\odot$ is said to be *round-to-nearest* if $\forall a, b \in R$

$$|(a \odot b) - (a \bullet b)| = \min_{x \in R}(x - (a \bullet b)).$$

Note that if a floating point operation is round-to-nearest, the result is the floating point number closest to the mathematically correct result. In case of a toss-up, i.e. if the mathematically correct result lies exactly between two floating point numbers, we accept either one. Another immediate consequence is that if the result of an operation is representable exactly by a floating point number, then we obtain the correct result without roundoff errors.

From the above definition, a bound for rounding errors and a useful condition for the mantissa of the result of a round-to-nearest operation $\odot$ easily follow. Let $z = m_z \cdot 2^{e_z} = a \odot b$. Then

$$|z - (a \bullet b)| < \epsilon_m \cdot z. \tag{2.1}$$

This is clear since if the error was more than $\epsilon_m \cdot z$ then either the floating point number $(m_z + 1) \cdot 2^{e_z}$ or $(m_z - 1) \cdot 2^{e_z}$ would be closer to the correct result. Furthermore for the mantissa m_z, the following equation holds.

$$m_z = \left[\frac{m_a \cdot 2^{e_a} \bullet m_b \cdot 2^{e_b}}{2^{e_z}} \right], \tag{2.2}$$

where $[x]$ denotes rounding to the nearest integer.

In most modern computers the constants $t, \underline{M}, \overline{M}$ are defined to follow the IEEE 754 standard[1]. The double precision numbers defined in that standard specify that $t = 53$, $\underline{M} = 1023$, $\overline{M} = -1024$. Thus, for double precision numbers $\epsilon_m = 2^{-53} \approx 10^{-16}$. Therefore in double precision we can represent about 16 valid decimal digits. The standard also defines that the elementary floating point operations $\oplus, \ominus, \otimes$ can be set to be round-to-nearest. Consistent with the notation introduced above, we will denote the set of double precision floating point numbers by R_{53}.

2.2 Exact operations. In the following subsections we will state some well-known facts about obtaining exact results for the basic floating point operations. While this may sound surprising at first, it is indeed possible to obtain the roundoff errors of the basic floating point operations exactly from within the floating point arithmetic. The theorems and proofs given here are originally due to Dekker[4], who showed that the theorems also hold with slightly lesser requirements on the underlying floating point operations than prescribed by the IEEE 754 standard. But since our implementation will build on IEEE 754 double precision floating point numbers, we will restrict ourselves to those. To give the reader an idea of how the proofs of those theorems work, we will prove some of the theorems while referring to [4] for others.

2.2.1 *Two-Sum.* The first theorem will provide us with a way to calculate the exact roundoff error occurring when adding two floating point numbers.

Theorem 2.3 *Let two double precision floating point numbers a and b such that $|a| > |b|$ be given. Let $z = a \oplus b$, $w = z \ominus a$ and $zz = b \ominus w$. Then, neglecting possible over- or underflows during the calculation, we have that $z + zz = a + b$ exactly.*

Proof Let $a = m_a \cdot 2^{e_a}$ and $b = m_b \cdot 2^{e_b}$. Since $|a| > |b|$ and floating point numbers are normalized, we have that $e_a \geqslant e_b$. It is sufficient to show that $w \in R_{53}$ and $b - w \in R_{53}$, then the result follows readily from optimality of the floating point operations.

Let $z = a \oplus b = m_z \cdot 2^{e_z}$. From Equation 2.2 we get that

$$m_z = \left[m_a \cdot 2^{e_a - e_z} + m_b \cdot 2^{e_b - e_z} \right].$$

Since $|a + b| < 2|a|$ we have that $e_z \leqslant e_a + 1$. Now we consider the two cases $e_z = e_a + 1$ and $e_z \leqslant e_a$.

- Assume $e_z = e_a + 1$. Then $m_z = \left[m_a \cdot 2^{-1} - m_y \cdot 2^{e_b - e_a - 1} \right]$ and letting $w = m_w \cdot 2^{e_a}$ we find that

$$\begin{aligned} |m_w| &= |m_z \cdot 2^{e_z - e_a} - m_a| \\ &= |m_z \cdot 2^{e_z - e_a} - m_a - m_b \cdot 2^{e_b - e_a} + m_b \cdot 2^{e_b - e_a}| \\ &\leqslant |2m_z - m_a - m_b \cdot 2^{e_b - e_a}| + |m_b \cdot 2^{e_b - e_a}| \\ &< 2|m_z - m_a \cdot 2^{-1} - m_b \cdot 2^{e_b - e_a - 1}| + 2^{53} \\ &< 2\frac{1}{2} + 2^{53}. \end{aligned}$$

  Since m_w is an integer, we therefore have that $m_w \leqslant 2^{53}$ and thus $w \in R_{53}$, i.e. w is a double precision floating point number.
- If $e_z \leqslant e_a$ the exact same proof carries through, the only difference being that we define $w = m_w \cdot 2^{e_z}$.

To prove that $zz \in R_{53}$, we first note that we can write $w = i \cdot 2^{e_b}$ for some integer i since $e_a \geqslant e_b$. Secondly, we have that $|b - w| = |b - z + a| \leqslant |b|$ by optimality. To see this simply let $z = x$, and then apply Definition 2.2. We thus have

$$|zz| = |b - w| = |m_b - i| \cdot 2^{e_b} \leqslant |b| = |m_b| \cdot 2^{e_b} < 2^{53} \cdot 2^{e_b},$$

and therefore $(m_b - i) \cdot 2^{e_b} = zz \in R_{53}$. □

Note that by Definition 2.1 floating point numbers are symmetric, i.e. if $a \in R$ then $-a \in R$. Thus the above theorem automatically provides exact subtraction as well.

It is worth mentioning that there are other algorithms to calculate the same two values without the condition that $a > b$, but requiring some additional floating point operations. The following algorithm is due to Knuth [6]. The advantage of this method is that due to pipelining on modern processors it is often faster to perform the three additional floating point operations instead of having to evaluate a conditional statement on the absolute values of a and b.

Theorem 2.4 *Let two double precision floating point numbers a and b be given. Let $z = a \oplus b$, $b_v = z \ominus a$, $a_v = z \ominus b_v$ and $zz = (a \ominus a_v) \oplus (b \ominus b_v)$. Then, neglecting possible over- or underflows during the calculation, we have that $z + zz = a + b$ exactly.*

Proof For a proof see, for example, [6]. $\square$

2.2.2 *Splitting.* Before we can move on to the exact multiplication, we introduce the concept of the splitting of a double precision number.

Definition 2.5 Let $a \in R_{53}$ be given. We call $a_h, a_t \in R_{26}$ the head and the tail of the splitting of a if

$$\begin{aligned} a_h &= \left[m_a \cdot 2^{-26}\right] \cdot 2^{e_x+26}, \\ a_t &= a - a_h. \end{aligned}$$

This definition may look surprising at first. After all a has 53 mantissa bits, but both a_h and a_t only have 26 bits each yielding a total of 52 bits. The solution to this riddle is the fact that the difference $|[x] - x| \leqslant 1/2$, but depending on x it can have either positive or negative sign. So the missing bit is the sign bit of the tail of the splitting.

Consider for example the number 7 in R_3. Its binary representation is given by $111 \cdot 2^0$, yet it can be split into two numbers in R_1 by $111 \cdot 2^0 = 1 \cdot 2^3 - 1 \cdot 2^0$.

The following theorem, also presented by Dekker, allows us to calculate such a splitting of a double precision number.

Theorem 2.6 *Let $a \in R_{53}$ be given and let $p = x \otimes (2^{27} + 1)$. Then the head of the splitting of a is given by $a_h = p \oplus (x \ominus p)$.*

Proof Since the proof of this theorem is somewhat technical and does not contribute much to the understanding of these operations, we refer the reader to the papers of Dekker[4] or Shewchuk[10]. $\square$

2.2.3 *Multiplication.* With the notion of a splitting, we can formulate the following theorem for exact multiplication of two double precision numbers:

Theorem 2.7 *Given two double precision floating point numbers a and b let $a = a_h + a_t$, $b = b_h + b_t$ be a splitting as defined above. Also let $p = (a_h \otimes b_h)$, $q = (a_t \otimes b_h) \oplus (a_h \otimes b_t)$ and $r = (a_t \otimes b_t)$. Then, neglecting possible over- or underflows during the calculation, $z = p \oplus q$ and $zz = (p \ominus z) \oplus q \oplus r$ satisfy $z + zz = a \cdot b$ exactly.*

Proof First note that for any two numbers $x, y \in R_{26}$ their product $x \cdot y \in R_{52} \subset R_{53}$. This is clear since for $x = m_x \cdot 2^{e_x}$ and $y = m_y \cdot 2^{e_y}$ we have that $x \cdot y = m_x \cdot m_y \cdot 2^{e_x+e_y}$ and $|m_x \cdot m_y| < 2^{52}$ since $|m_x| < 2^{26}$ and $|m_y| < 2^{26}$.

We also have that

$$a \cdot b = (a_h + a_t) \cdot (b_h + b_t) = a_h \cdot b_h + a_h \cdot b_t + a_t \cdot b_h + a_t \cdot b_t.$$

Since $a_h, a_t, b_h, b_t \in R_{26}$, each single term in this sum is in R_{52}. Furthermore, the two cross terms $a_h \cdot b_t$ and $a_t \cdot b_h$ have the same exponent and therefore their sum is in R_{53}. Thus p, q, and r, as defined in the statement of the theorem, are exact, and we obtain that $a \cdot b = p + q + r$.

Now we perform an exact addition of p and q as described above, yielding the leading term $z = p \oplus q$ and a remainder term $z_1 = (p \ominus z) \oplus q$. We thus have $a \cdot b = z + z_1 + r$. Close examination of the proof of the exact addition shows that r and z_1 have the same exponent and both are in R_{52}, so their sum can be calculated exactly in R_{53}. This leaves us with the final equation $a \cdot b = z + (z_1 \oplus r) = z + ((p \ominus z) \oplus q \oplus r) = z + zz$, which completes the proof. $\square$

3 High precision numbers

Based on the exact multiplication and addition presented in the previous section, it is now possible to implement high precision numbers. A high precision number is stored as an unevaluated sum of double precision floating point numbers. The value represented by that high precision number is given by the mathematically exact sum of all terms:

Definition 3.1 A high precision number a is given by a finite sequence of double precision floating point numbers a_i. We call each a_i a limb of the number. The value of a is given by

$$a = \sum_{i=1}^{n} a_i.$$

The sequence a_i is also called a floating point expansion of a.

Note that in this definition we do not specify any requirements as to the relative size of the a_i. In general we would like the a_i to be ordered by magnitude in such a way that $|a_i| \approx \epsilon_m |a_{i-1}|$. If that condition is true for all limbs, we call the number *normalized.*

Depending on the desired accuracy, the maximum length of the expansion is fixed before calculations commence. Although the machine precision is almost 10^{-16}, we conservatively estimate that each additional limb adds 15 more significant decimal digits to the expansion. Thus for a desired accuracy of n digits, the number of limbs necessary is given by $\lceil n/15 \rceil$.

In order to turn the high precision numbers rigorous, we add an error bound to the expansion, similar to the remainder bound of Taylor Models[7].

Definition 3.2 A high precision interval a is given by a high precision number consisting of n limbs a_i and a double precision error term a_{err}. The value of the interval is then given by

$$a = \left[\sum_{i=1}^{n} a_i - a_{err}, \ \sum_{i=1}^{n} a_i + a_{err} \right].$$

For shorter notation we also denote the above interval by $a = \sum_{i=1}^{n} a_i \pm a_{err}$.

This way of storing intervals as only one high precision midpoint and a simple double precision error term has obvious advantages over intervals stored as two high precision endpoints. Only one high precision number is needed, so the memory footprint of the high precision intervals is smaller. Furthermore, the computation time is less since operations only need to operate on one high precision number, whereas the error term can be calculated quickly in double precision arithmetic. Finally, this representation fits in nicely with the general concept of our high precision numbers. As we will see in the next section, verification is almost automatic in our algorithms. Thus our high precision intervals are almost as fast as non-verified high precision numbers would be.

Note that there are several other implementations of non-verified high precision numbers based on the concept of floating point expansions (e.g. [10]). To the best of our knowledge, however, our implementation is the only rigorous interval implementation based on this concept.

3.1 Addition. The most elementary operation to be implemented for two high precision intervals is addition. Given two high precision intervals $A = \sum_{i=1}^{n} a_i \pm a_{err}$ and $B = \sum_{i=1}^{n} b_i \pm b_{err}$, the exact sum is given by

$$A + B = \sum_{i=1}^{n} a_i + \sum_{i=1}^{n} b_i \pm (a_{err} + b_{err})$$

While this is a valid expansion with $2n$ limbs, it is not a valid n limb expansion. Thus the addition operation can be reduced to the problem of accumulating a sequence of m_1 floating point numbers into a sequence of $m_2 < m_1$ floating point numbers.

The algorithm to perform that operation is called the accumulator. It takes a sequence of n double precision numbers a_i and returns another sequence of double precision numbers b_i of predefined maximum length. If there are roundoff errors, or the result does not fit into the requested length, the remaining terms are accumulated in an outward rounded error term. This error term represents an upper bound on the error of the result.

The implementation of this accumulator algorithm is not complicated. Let $a_1, \ldots, a_n$ denote the double precision numbers in the input array. Using the exact addition presented in the previous section, we begin by adding a_1 and a_2 exactly resulting in a result sum_1 and an error term b_1. Then we continue to exactly add sum_1 and a_2 into sum_1 and an error term b_2. This process is repeated until we have added all a_n. The resulting term sum_1 then is the first limb of the result. Note that after this procedure we are left with $b_1, \ldots, b_{n-1}$ error terms. To calculate the next limb, we just repeat the same procedure on $b_1, \ldots, b_{n-1}$, and so forth. Once the maximum number of limbs is reached, the absolute values of all remaining error terms are added up and rounded outwards to give a rigorous bound on the error. This algorithm is graphically represented in Figure 1.

Note that this implementation of the addition does not pose any restrictions on the input sequence. In particular, it is not required to be in any specific order or even normalized, like many other algorithms for floating point expansions require. To minimize the roundoff errors, speed up execution time, and obtain optimally sharp results, it is best if the input is sorted by decreasing order of magnitude. But even with completely random input, the result will be fully rigorous nonetheless.

The output sequence also is not guaranteed to be ordered, yet typically it will even be normalized. The extend of the denormalization of the result depends on the amount of cancellation happening during the additions.

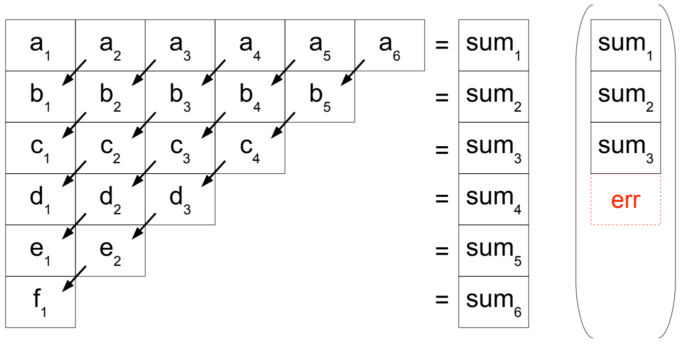

Figure 1 The accumulator algorithm. In this example, six double precision numbers $a_1 \dots a_6$ are to be added. The arrows indicate the error term of the exact addition of the two numbers above them. If only three limbs are desired for the result, the summation terminates after three iterations and the left over terms are accumulated into an outward rounded error term.

3.2 Multiplication. The second most important operation is multiplication. Given two high precision intervals $A = \sum_{i=1}^{n} a_i \pm a_{err}$ and $B = \sum_{i=1}^{n} b_i \pm b_{err}$, the exact product is given by

$$\begin{aligned} A * B &= \left(\sum_{i=1}^{n} a_i\right) * \left(\sum_{i=1}^{n} b_i\right) \pm \left(a_{err} * \left|\sum_{i=1}^{n} b_i\right| + b_{err} * \left|\sum_{i=1}^{n} a_i\right| + a_{err} * b_{err}\right) \\ &\subset \sum_{i,j=1}^{n} a_i * b_j \pm \left(a_{err} * \left(\sum_{i=1}^{n} |b_i| + b_{err}\right) + b_{err} * \left(\sum_{i=1}^{n} |a_i| + a_{err}\right)\right) \end{aligned}$$

Thus, multiplication essentially reduces to multiplying each limb of the first argument with each limb of the second argument using Dekker's algorithm, and summing up the resulting floating point numbers. The error term given above can easily be bounded from above using outward rounding floating point arithmetic, since all terms summed up are positive.

While the accumulator could be used to achieve this goal, it is possible to utilize the additional information about the structure of the high precision intervals to speed up the multiplication without losing the correctness of the result. Assuming the input numbers are normalized, it is possible to estimate which is the first limb of the result affected by adding the product $a_i * b_j$ to the result. Under these assumptions, the product $a_i * b_j$ is of order of magnitude $\epsilon^{i+j-2}(a_1 * b_1)$. Assuming the result C is also normalized, the first limb affected by adding $a_i * b_j$ is c_{i+j-1}.

When performing the addition, all limbs before c_{i+j-1} remain unchanged, and thus the addition can be started at this limb. The exact multiplication of a_i and b_i yields two floating point numbers $e_{1,1}$ and $e_{1,2}$. To add these to the result, two exact additions are performed. First $e_{1,1}$ and c_{i+j-1} are added exactly into c_{i+j-1} and $e_{2,1}$. Then c_{i+j-1} and $e_{1,2}$ are added exactly into c_{i+j-1} and $e_{2,2}$. The two newly created error terms $e_{2,1}$ and $e_{2,2}$ are of order $\epsilon * e_{1,1}$ and thus adding them to the result C will only affect limbs starting at c_{i+j}.

The same process is repeated $k-1$ times, until the last limb of the result has been reached. At this point, $e_{k,1}$ and $e_{k,2}$ contain the exact errors that are left over in the addition. Since there are no more limbs in the result to hold these numbers,

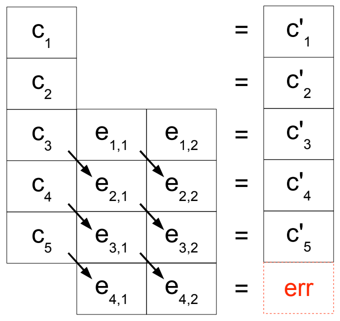

Figure 2 The partial addition in the multiplication algorithm. In this example, $e_{1,1}$ and $e_{2,1}$ are added to the result $c_1 \ldots c_5$ starting at the third limb c_3. The arrows indicate the error term of the exact addition of the two numbers above them. $e_{4,1}$ and $e_{4,2}$ are added to the error term of the result.

their absolute values are simply added to the remainder bound. This process yields a fully rigorous addition of the product $a_i * b_j$ to the result C (see Figure 2).

Note that for products that are determined not to affect any limbs in the result, no exact multiplication has to be performed, instead an outward rounded floating point multiplication is sufficient to determine the value to be added to the error bound directly.

3.3 Division. Based on the the multiplication and addition described above, it is now possible to implement division and a square root. For the division, a constructive approach is used, that produces a better result by adding correction terms limb by limb.

The algorithm is basically the same as the simple "black board" division (also called long division), that is taught in middle school. It makes use of the division for floating point numbers to estimate the next limb in the result. Then rigorous multiplication and subtraction are used to calculate how far off the new result is from the correct result.

Given A and B, $C = A/B$ is to be computed. First, A and B are approximated by floating point numbers a and b. Those approximate numbers are then divided using floating point arithmetic to yield $c = a/b$. The result c is then used as a first approximation of the high precision result, serving as its first limb c_1.

In the next step, the high precision interval operations are used to rigorously calculate the difference $D_1 = A - B * c_1$, i.e. the error left to correct. Letting d_1 be a floating point approximation of D_1, the same process yields an approximation of the next limb $c_2 = d_1/b$. In general, letting $D_k = D_{k-1} - B * c_k$, one inductively obtains as many limbs c_k as required. This process also readily yields a rigorous error bound in D_k.

3.4 Square Root. To compute the square root X of a high precision number A, we use the interval Newton method[8] to obtain an interval enclosure of the solution of $X^2 - A = 0$.

The iteration step for this equation is given by

$$X_{n+1} = M(X_n) - \frac{M(X_n)^2 - A}{2X_n}$$

where $M(X_n)$ stands for the midpoint of the interval X_n.

Provided that the initial interval enclosure X_0 contains the correct square root interval, each subsequent X_n contains the correct result[8]. Furthermore, the sequence of intervals converges quickly. It is not necessary to intersect each X_n and X_{n+1}, as this operation is expensive in terms of computational cost. The criterion for terminating the iteration is the size of the error bound of X_n. Once the size of the error bound grows, i.e. the intervals do not contract any more, the best possible enclosure of the result has been reached.

The initial starting value X_0 is chosen by double precision interval arithmetic. First, an interval enclosure I of the high precision interval A is calculated. If it exists, the interval square root of I is converted into a high precision number, which serves as the initial interval X_0. If the interval I contains negative numbers, an error flag in the result X is set instead.

Note that in our representation of high precision intervals, it is particularly easy to obtain an the midpoint of an interval. All that has to be done is to set the error bound to 0 temporarily. This way, all further calculations are still performed in rigorous interval arithmetic, thus assuring that the resulting interval X_{n+1} really contains the correct result.

3.5 Intrinsic Functions. All other intrinsic functions are implemented based on their Taylor expansions and using identities relating the different intrinsic functions to each other. Typically, the argument is reduced before being passed to the Taylor series, to speed up convergence. These argument reductions are also based on mathematical identities, very much depending on the intrinsic function[2].

The verification of the result is very much dependent on the specific intrinsic function as well. Often it is possible to use the Lagrange remainder formula to obtain a rigorous upper bound of the remainder error. In other instances it is possible to use other remainder criteria, such as the Leibniz criterion.

In this paper, we will present the implementation of the arcus tangent function. Other intrinsics are implemented similarly, for a detailed discussion see[12].

For the arcus tangent, the following expansion is used[3]:

$$\arctan X = \sum_{n=0}^{\infty}(-1)^n \frac{X^{2n+1}}{2n+1} = X - \frac{1}{3}X^3 + \frac{1}{5}X^5 + \dots$$

This series converges for $|X| \leqslant 1$. To reduce the argument to that range, the following identity is used[3]:

$$\arctan(X) = 2\arctan\frac{X}{1+\sqrt{1+X^2}}$$

Applying this equation once, reduces all arguments from $(-\infty, +\infty)$ to $(-1, 1)$. Because convergence of the series for that range still is slow, the reduction is applied two more times. This reduces the argument to the interval $(-0.25, 0.25)$. After evaluating the series for the reduced argument, only a multiplication by 8 is required. This can be performed exactly by multiplying each limb and the error term in floating point arithmetic.

To obtain an error bound on the remainder of the series, it is in this case not possible to use the Taylor remainder formula. Careful examination of the term $\sup_{x\in(-0.25,0.25)}\left(\frac{d}{dx}\right)^n \arctan x$ shows, that it grows quickly, so that the Lagrange remainder term only converges as $\frac{1}{n}$. This overestimates the error by several orders of magnitude. Fortunately, the series has monotonously decreasing elements and alternating signs. Thus the Leibniz criterion[3] applies, and an estimate of the remainder to the partial series is given simply by the next term in the series:

$$R_n = \left| \frac{X^{2n+3}}{2n+3} \right|$$

3.6 Comparison to other packages. It is worth noting some fundamental differences between our implementation and other high precision implementations based on floating point expansions, such as the "double double" implementation in the original Dekker paper [4] and the arbitrary precision arithmetic by Shewchuk [10].

In those implementations the authors were very careful to develop algorithms that provide some guaranteed accuracy and derived error estimates for their operations. For floating point expansions, these estimates always lead to a requirement for normalization of the input. Normalization, however, is a rather complicated and computationally expensive operation.

We, on the other hand, use an entirely different approach. Instead of making any analytic estimates for roundoff errors, we have our numbers validate themselves by adding a rigorous remainder bound. The only claim we make is that the interval resulting from an operation rigorously encloses the mathematically correct result. If the input numbers are denormalized, or if excessive cancellation occurs during the calculation, it is possible that our algorithm produces significant overestimation. The result, however, is rigorous nonetheless. Applying our high precision intervals to real world problems shows that those cases are very rare and that our intervals typically provide sharp enclosures of the correct results.

We also compared the speed of our implementation to the speed of MPFI. For this comparison, we added an MPFI data type to COSY INFINITY. First, the same intervals are initialized once as the MPFI data type and once as our high precision intervals. The same operation is then repeated 10.000 times at different precisions, and the CPU time taken for all operations is measured.

Representative for the other operations, Figures 3 and 4 show the ratio between the runtime for our implementation and the runtime for MPFI for multiplication and the arcus tangens intrinsic function respectively. The runtime ratio HI/MPFI is plotted as a function of the number of limbs. The range of limbs tested lies between 1 and 5, corresponding to a range of 15 to 75 significant decimal digits.

For multiplication, our code performs faster over the entire range of precision. Even at 75 digits, our algorithms still only take about 83% of the time MPFI takes to multiply two intervals. Similar results are true for addition and subtraction, where our algorithms outperform MPFI by at least a factor of two.

For more complicated operations, such as certain intrinsics, MPFI outperforms our implementation at 75 digits by up to a factor of 3. This is probably due to highly optimized algorithms for evaluating intrinsic functions. At 45 significant digits, however, our implementation still is en par with MPFI for most intrinsics.

At higher precisions, our code performs significantly worse. However, our goal in this implementation was to implement a portable, fast and rigorous interval package for precisions up to about 100 significant digits. In particular, we focused on the elementary functions such as addition and multiplication, as these are most commonly used.

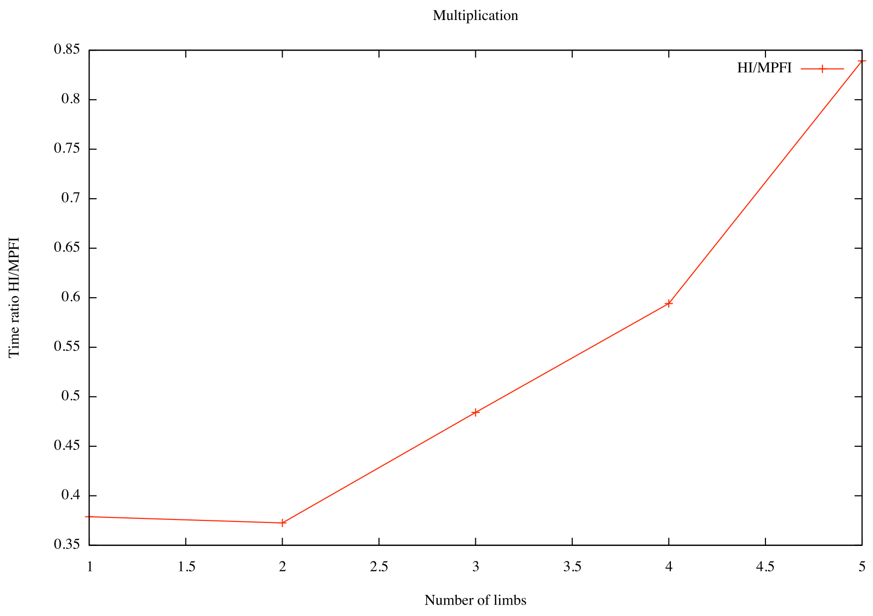

Figure 3 Speed comparison between our interval implementation and MPFI for multiplication.

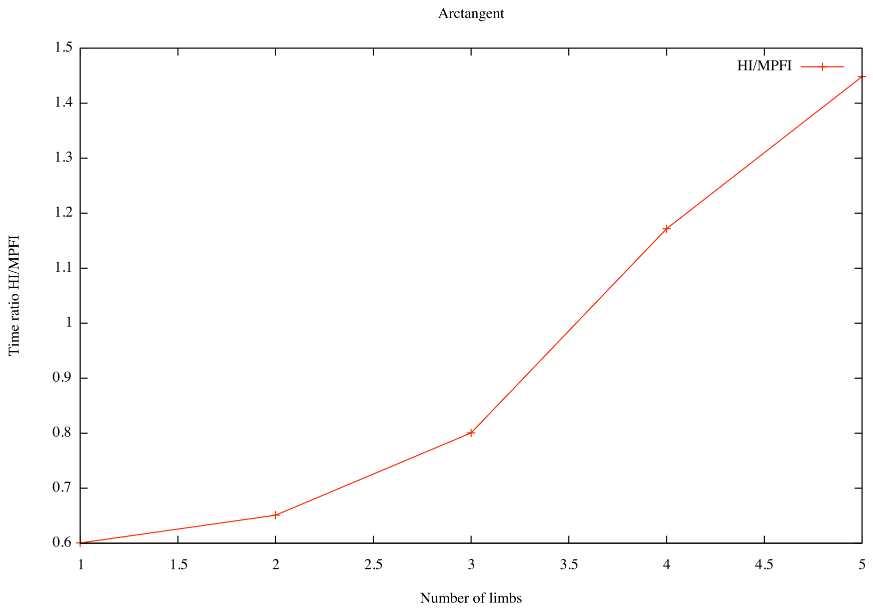

Figure 4 Speed comparison between our interval implementation and MPFI for the arcus tangens intrinsic function.

4 Outlook

Our high precision intervals have already proved useful in applied rigorous numerical computations. Among other things, we used our high precision intervals to compute a verified high precision enclosure of an attractive fixed point in the Hénon map[13].

Furthermore, based on the algorithms presented in this paper, we implemented non-verified, high precision differential algebra vectors[11]. These are a first step towards high precision Taylor Models, which are being implemented right now. In particular, the intrinsic functions for high precision intervals will be very useful in the implementation of the corresponding intrinsics for high precision Taylor Models.

References

[1] IEEE standard for binary floating-point arithmetic. Technical Report IEEE Std 754-1985, The Institute of Electrical and Electronics Engineers, 1985.

[2] K. Braune and W. Kramer. High-accuracy standard functions for intervals. In Manfred Ruschitzka, editor, *Computer Systems: Performance and Simulation*, volume 2, pages 341–347. Elsevier Science Publishers B.V., 1986.

[3] Bronstein, Semendjajew, Musiol, and Mühlig. *Taschenbuch der Mathematik*. Verlag Harri Deutsch, 5th edition edition, 2001.

[4] T.J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 1971/72.

[5] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software*, 33(2):13:1–13:15, 2007.

[6] D. E. Knuth. *The Art of Computer Programming*, volume I-III. Addison Wesley, Reading, MA, 1973.

[7] K. Makino. *Rigorous Analysis of Nonlinear Motion in Particle Accelerators*. PhD thesis, Michigan State University, East Lansing, Michigan, USA, 1998. Also MSUCL-1093.

[8] R. E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1966.

[9] N. Revol. Multiple precision floating-point interval library. Technical report, SPACES, INRIA Lorraine and Arenaire, INRIA Rhone-Alpes, 2002.

[10] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18:305–363, 1997.

[11] A. Wittig and M. Berz. Computation of high-order maps to multiple machine precision. *International Journal Modern Physics A*, in print, 2008.

[12] A. Wittig and M. Berz. Design and implementation of a high precision arithmetic with rigorous error bounds. Technical Report MSUHEP 081126, Michigan State University, December 2008.

[13] A. Wittig, M. Berz, and S. Newhouse. Computer assisted proof of the existence of high period periodic points. *Communications of the Fields Institute*, in print, 2008.