Design and Implementation of a High Precision Arithmetic with Rigorous Error Bounds

Alexander Wittig
MSUHEP-081126

December 2008

In this paper we present the design of a rigorous, high precision floating point arithmetic. The algorithms presented are implementation in FORTRAN, and are made available through the COSY INFINITY rigorous computation package.

The three design objectives for these high precision intervals are high speed, particularly for the elementary operations, absolutely rigorous treatment of round-off errors, and scalability to arbitrary precision. The main focus of these algorithms lies on relatively low precision of up to 100 significant digits.

Unlike many available high precision packages, we do not rely on large integer arithmetic to emulate floating point numbers with arbitrary mantissa length. Instead, we utilize unevaluated series of double precision floating point numbers. Combined with algorithms for exact operations on floating point numbers, this allows us to implement operations very efficiently.

Contents

1 Introduction

In this paper we design a high precision library that allows calculations with typically about three to four times the precision of regular double precision numbers. While in principle the achievable precision with this implementation is much higher, we will focus on these "low" higher precision numbers to be fast and efficient.

Our main objectives for the design, as well as the implementation of the algorithms are

Mathematical Rigor The algorithms have to be absolutely rigorous. That is, in addition to the high precision representation of the number itself, we will keep an error bound. The actual number to be represented is guaranteed to be included in this error bound. The error bound has to be kept current throughout all operations performed by our algorithms. It includes all possible sources of errors, including roundoff errors due to floating point operations.

Speed Since the algorithms are included in the COSY [9] software package, they have to perform well speed-wise. We will optimize our algorithms, using the fact that our calculations do not have to be absolutely precise, as long as we make them rigorous.

Arbitrary length While focusing on the range of up to about 100 significant digits, the code is planned such that it is not restricted to quadruple or any other fixed precision.

1.1 Floating Point Numbers

To represent calculations on the real numbers on a computer, most modern processors use floating point numbers. The concept behind the representation of a floating point number is essentially the same as the "scientific notation" in terms of relevant digits and a power of ten to represent the order of magnitude. The same is done with floating point numbers. The only difference is that, due to the binary number system, a power of two is used to signify the magnitude.

Definition 1. *We define the set of all floating point numbers R to be given by*

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

where m_z and e_z are integers.

The constants $t, \underline{M}, \overline{M}$ are fixed integers and define the floating point number system. $\underline{M}$ and $\overline{M}$ define 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 of the exponents 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 1 that

$$\frac{|r - \tilde{r}|}{|r|} \leqslant 2^{-t} = \varepsilon_m/2$$

The value $\varepsilon_m = 2^{-t+1}$ 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. *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 is exactly between two floating point numbers, we will 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 this 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:

$$|z - (a \bullet b)| \leqslant \varepsilon_m/2 \cdot |z| \tag{1}$$

This is clear since if the error was more than $\varepsilon_m/2 \cdot |z|$ 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:

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

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 [2]. The double precision numbers defined in that standard specify that $t = 53$,

$\underline{M} = 1023$, $\overline{M} = -1024$. Thus, for double precision numbers $\varepsilon_m = 2^{-52} \approx 2 \cdot 10^{-16}$. 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}.

1.2 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 [5], 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 the reader to [5] for others.

1.2.1 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 1. *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.*
Furthermore, in round-to-nearest double precision arithmetic $|zz| \leqslant 2^{-53} \cdot |z|$.

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 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 $m_z = \left[m_a \cdot 2^{-1} - m_b \cdot 2^{e_b - e_a - 1}\right]$ and letting $w = m_w \cdot 2^{e_a}$ we find that

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

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

The estimate $|zz| \leqslant 2^{-53} \cdot |z|$ follows directly from Equation 1. Since z is the result of a round-to-nearest operation, it is accurate up to an error of less than $|z| \cdot \varepsilon_m/2$, which is precisely given by zz. □

Note that by Definition 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 also other algorithms to calculate the exact same two values without the condition that $a > b$, but requiring some additional floating point operations. The following algorithm is due to Knuth [8]. 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. *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, [8]. □

1.2.2 Splitting

Before we can move on to the exact multiplication, we require the concept of the splitting of a double precision number.

Definition 3. *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*

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

This definition may sound 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 $|[a] - a| \leqslant 1/2$, 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.

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

Theorem 3. *Let $a \in R_{53}$ be given and let $p = a \otimes (2^{27} + 1)$. Then the head of the splitting of a is given by $a_h = p \oplus (a \ominus p)$.*

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 [5] or Shewchuk [12]. □

1.2.3 Multiplication

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

Theorem 4. *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. □

For the size of the correction term zz the following estimate is given by Dekker (see [5]) for standard double precision (R_{53}):

$$|zz| \leqslant |a \times b| \frac{3 \cdot 2^{-53}}{1 + 3 \cdot 2^{-53}} < |x \times y| \cdot 3\varepsilon_m/2$$

where $a \times b$ is the correct mathematical result.

1.3 Kahan Summation

The Kahan summation algorithm [7] is an algorithm, that allows the summation of a sequence of floating point numbers, producing a much smaller error than a naive floating point

summation would yield. It follows the idea of the Dekker addition, by calculating the round-off error in each addition and feeding it back into the calculation in the next step. While the result is, of course, still not exact, the roundoff error is greatly reduced compared to naive addition.

Let sum be the floating point sum of a sequence of floating point numbers a_i, and $c = 0$ initially. Then, for each a_i the following operations are preformed in floating point arithmetic.

$$\begin{aligned} y &= a_i - c \\ t &= sum + y \\ c &= (t - sum) - y \\ sum &= t \end{aligned}$$

We will not go into details of the algorithm here, for more information see [6, 7].

1.4 Newton Method

The Newton Method is a well known method for iteratively finding the zeros of a function $f(x)$ [4]. For all sufficiently smooth functions f, there is a neighborhood of the solution, in which the convergence of this method is super-linear. This is of particular interest for our problems, since we can use the available non-rigorous floating point intrinsics to quickly obtain an approximation of the result, which we then can use as a starting value. From there, it typically takes very few iterations to converge to the final result in the desired precision.

The Newton iteration step is calculated according to the equation

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{3}$$

1.5 Interval Newton Method

Based on the traditional Newton method, Moore introduced the following, slightly modified, interval Newton method [10].

$$X_{n+1} = m(X_n) - \frac{f(m(X_n))}{f'(X_n)} \tag{4}$$

Here $m(X_n)$ stands for the midpoint of the interval X_n.

If all operations in the above equation are carried out rigorously, and the initial interval contained a zero of the function f, the resulting interval X_{n+1} is also guaranteed to contain a zero. Thus, by iterating Equation 4, one can obtain an interval enclosure of a zero of f. For this method to work, it is required that the interval Newton step actually contracts the argument.

Note that it is possible to define X_{n+1} as the intersection of X_n and the interval Newton step on acting X_n. The above statements still hold in that case. This intersection, however, often is not necessary if the interval Newton step is contracting the interval enough.

1.6 Taylor Expansion

Using the Taylor expansion to evaluate certain intrinsic functions is a very powerful method to obtain good interval approximations [3, 4]. In particular, in many cases we can derive an analytic expression for the remainder term.

A Taylor expansion of a function $f(x)$ up to order n (around 0) is given by Equation 5

$$f(x) = \sum_{i=0}^{n} \frac{f^i(0)}{i!} x^i + R_n \tag{5}$$

The remainder term of that expansion in the Lagrange form is given by:

$$R_n = \frac{f^{n+1}(\xi)}{(n+1)!} x^{n+1} \tag{6}$$

for some $\xi \in (0, x)$. While there are other forms for the remainder term, this one is the most useful for our purposes.

Given that $||f^{n+1}||_\infty$ exists, i.e. f^{n+1} is bounded over the whole domain of x, we can simplify the remainder term as follows:

$$|R_n| \leqslant \frac{||f^{n+1}||_\infty}{(n+1)!} |x|^{n+1} \tag{7}$$

1.7 Leibniz Criterion

Another method for bounding the remainder of a sum is the Libniz criterion [4]. Given a monotonically decreasing sequence a_n such that $\lim_{n\to\infty} a_n = 0$. Then $a_n \geqslant 0$ for all n and the series

$$s = \sum_{n=0}^{\infty} (-1)^n a_n$$

does converge. Furthermore, the remainder after the mth partial sum is given by

$$|s_m - s| = \left| \sum_{n=0}^{m} (-1)^n a_n - \sum_{n=0}^{\infty} (-1)^n a_n \right| = \left| \sum_{n=m+1}^{\infty} (-1)^n a_n \right| \leqslant |a_{m+1}|$$

2 Algorithms

In this section, we present the detailed algorithms used in our high precision algebra. We will ignore certain purely implementatory aspects, those will be discussed section 3.

2.1 Representation

Definition 4. *A high precision interval A is given by a finite sequence of double precision floating point numbers 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 shorthand notation, we will also refer to the interval as $A = \sum_{i=1}^{n} a_i \pm a_{err}$.

We call a_i the i-th limb of the interval, the number of limbs n is its length. We call the exact sum of all limbs the value of the high precision interval. The rigorous remainder a_{err} is another floating point number that specifies the uncertainty (or width) of the high precision interval. The correct number represented by A is guaranteed to be in the interval $[\sum a_i - a_{err}, \sum a_i + a_{err}]$. We require a_{err} to always be a positive number. The sequence a_i is also called a floating point expansion of the midpoint of A.

A high precision interval A is called *well conditioned* if for all i we have $|a_i| > |a_{i+1}|$, i.e. the absolute values of the sequence are strictly decreasing.

A high precision interval A is called *normalized* or *canonical* if for all i we have $\varepsilon \cdot |a_i| > |a_{i+1}|$, i.e. the floating point numbers are non-overlapping where ε is the machine precision. Obviously normalized numbers are also well conditioned.

2.1.1 Limitations

This representation implies certain limitations to the range of our high precision intervals. Those limitations are a direct consequence of the limitations of double precision floating point numbers.

The range of representable numbers is obviously limited by the range of double precision numbers. The smallest possible non-zero number representable consists of one single limb representing the smallest possible double precision floating point number. That number is of order $2^{-1024} \approx 10^{-309}$.

The largest representable number, in theory, is only limited by available memory, since there is always an unbounded sequence of double precision floating point numbers. In practice, however, we would like our numbers to be normalized. Therefore, the largest representable number is of order $2^{1024} \approx 10^{309}$. Furthermore, our algorithms will encounter over- and underflow errors during the calculations when limbs get close to the double precision limits.

The relative precision of our high precision intervals is bound by the "quantization" in quanta of the size of the smallest possible representable double precision number. Thus, the last digit in a decimal representation of a number is of order 10^{-309}. A number of order one can, therefore, have a maximum of about 309 valid decimal digits. Note that the maximum number of valid digits depends on the size of the represented number itself.

Thus, we have the following limitations for a high precision number X:

$$10^{-309} \lesssim |X| \lesssim 10^{309} \tag{8}$$

where the relative precision depends in the order of X.

2.2 The Accumulator

The key operation in our algorithms is the accumulator. Every other operation will, directly or indirectly, call this function. The accumulator simply takes an arbitrary list of floating point numbers, and adds them up exactly using Dekker's addition. The result will be a high precision interval of a specified maximum length, that represents the sum of the input rigorously.

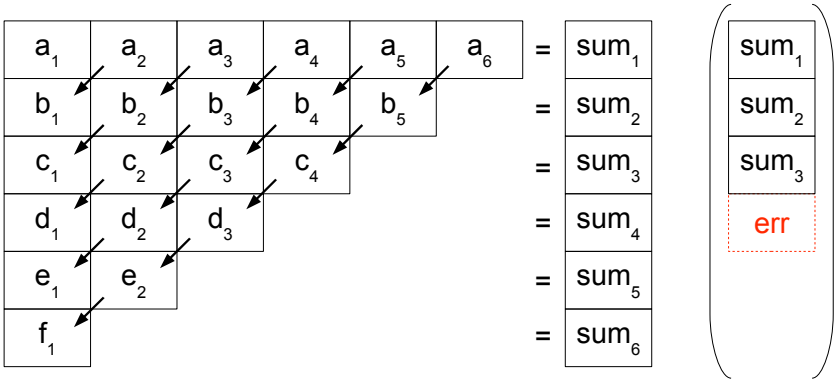

Figure 1: Graphical representation of the accumulator algorithm for $n = 6$. There are two cases shown: In the first the result holds six limbs. In the second case, shown in parentheses, the result holds only three limbs. In that case, the accumulation terminates after calculation three limbs. The absolute values of the remaining terms d_i are then added to the error bound.

If the length of the output interval is not sufficient to hold the exact result, the accumulator adds the leftover numbers to the error bound of the result. Consequently, the result is always a rigorous enclosure of the mathematically correct result. For performance reason, we also add terms to the error bound, that are smaller than the error bound itself. This prevents computationally expensive calculations on numbers that are already below the remainder term. For this purpose, the initial error bound of the result is passed to the accumulator. This allows us to first calculate the error bound of the resulting interval of an operation, and then pass it to the accumulator.

Let a_i denote the input sequence with n elements, and S the resulting high precision interval of maximal length m. The accumulator then sums up the a_i successively using $n-1$ exact additions. This results in one double precision number representing an approximate result of the final sum. This number becomes the first first limb s_1 of the result. In the exact additions we also obtained $n-1$ roundoff terms. To these, the same scheme is applied again to calculate the second limb of the result. This calculation is repeated until there are either no more roundoff errors left, or the maximum number of limbs in the result has been reached. If there are terms left, the absolute values of those terms are added to the result's error term s_{err}. For $n = 6$ and $m = 3$ this process is graphically shown in Figure 1.

To increase performance, we add a slight adjustment to this algorithm. After performing each exact addition, we check if the roundoff error of the result actually is above s_{err}. Only in that is the case is it kept for the next round, otherwise the roundoff error is immediately added to s_{err}. The same test is applied to the resulting limb in each step. If its absolute value is less than the error bound, it is also added to s_{err} and the limb is discarded. The only exception is the very first limb, which is always kept, in order to prevent large intervals from always being centered at 0.

It is important to note that the order of the input numbers is important. It does influence the result and, more importantly, the performance of this algorithm. Best performance is obtained if the sequence a_i is ordered by increasing absolute value of the numbers. If two small numbers just above the error bound are added, the absolute value of the roundoff error for that operation will surely be below the error bound. As described above, it will therefore immediately be discarded from the further calculation. If, however, one such small number is added to an already significantly larger number, the roundoff will be of the order of the small number added. Thus the small number will just propagate as a roundoff error into the next round. So for best performance, it is important that, if possible, the numbers are at least roughly ordered by magnitude.

It is also interesting to note that the output of this algorithm is not necessarily normalized. In fact, if cancellation occurs in the addition, this can lead to poorly conditioned results. However, in most real world cases it is very likely, that the numbers returned are well conditioned. This possibility for cancellation is the reason why none of our algorithms have any normalization requirements for the input numbers. However, we do optimize our algorithms to perform better with well conditioned or even normalized input.

2.3 Addition and Subtraction

Given the accumulator, addition of two high precision intervals A and B into C is quite trivial. All that has to be done is to concatenate the sequences of limbs making up the two numbers into a temporary array, and apply the accumulator to it. Before calling the accumulator, the two error terms of A and B are added up to form the new error term $c_{err} = a_{err} + b_{err}$. This new error term is then passed to the accumulator, along with the temporary array.

As mentioned above, passing roughly sorted input to the accumulator improves performance. So, instead of blindly concatenating the two sequences a_i and b_i, it is possible to merge them in order, assuming the high precision intervals A and B themselves are well conditioned. This is achieved by starting at the last limb of both sequences and successively copying the smaller of the two numbers into the temporary array. This yields, in linear time, a sequence of numbers ordered by increasing absolute value. If the input is not well conditioned, best performance of the accumulator would be obtained by a full sort of the input. In practice, however, the performance gain in the accumulator is too small to outweigh the cost of sorting the input.

For the subtraction $A - B$, the same algorithm as for addition is used, except all signs of the b_i are flipped. Since the set of floating point numbers is symmetric, i.e. if a is a floating point number so is $-a$, this operation is exact.

2.4 Multiplication

The multiplication is the core of our algorithms. Together with addition, this operation will be the most important and the most frequently performed operation. It is, therefore, vital to make this operation as efficient as possible.

Mathematically, it is clear what has to be done. To multiply two high precision numbers A and B one has to perform the following operation:

$$C = \left(\sum_n a_n\right) \cdot \left(\sum_m b_m\right) = \sum_{m,n} a_n b_m$$

It is obvious that, in general, there are $n \cdot m$ multiplications necessary to evaluate this expression. Each of these multiplications has to be done exactly, using a Dekker multiplication, and yields two resulting terms. Thus, in the end there are $2 \cdot n \cdot m$ terms to be summed up using the accumulator (subsection 2.2).

However, in practice it is possible to reduce the number of multiplications needed significantly. Once again this is due to the size of the error bound. The new error bound c_{err} is given by

$$\begin{aligned} c_{err} &= \left(\sum_n a_n\right) \cdot b_{err} + \left(\sum_m b_m\right) \cdot a_{err} + a_{err} \cdot b_{err} \\ &\leqslant \left(\sum_n |a_n| + a_{err}\right) \cdot b_{err} + \left(\sum_m |b_m| + b_{err}\right) \cdot a_{err} \end{aligned}$$

As before, this new error bound is calculated first. Then, before performing a full Dekker multiplication, a simple floating point multiplication $x = a_i \cdot b_j$ of the two limbs a_i and b_j is performed. This product is then compared to c_{err}. There are three possible cases:

$|x| < c_{err}$ In this case, the result of the multiplication is smaller than the error term. Its absolute value can thus be rounded outwards and added to the error bound. No further processing is necessary.

$\varepsilon \cdot |x| < c_{err}$ In this case, the result of the multiplication does contribute to the final sum significantly. The roundoff error of the floating point multiplication, however, is below the error bound and thus can be estimated from above as $\varepsilon \cdot x$. The absolute value of this is then added to the error bound. Note that this works, as long as we have round-to-nearest operations.

$Else$ In all other cases, a full Dekker multiplication has to be performed. The result of this multiplication will consist of two floating point numbers, which have to be stored in a temporary array. This will eventually be handed off to the accumulator later. We also check wether the smaller of the two resulting numbers if it is below the error term. If so, we add the number to the error bound directly, thus reducing the number of elements, that are passed to the accumulator.

This way, we can reduce the number of Dekker multiplications, that are actually performed, significantly. It seems this results in an improvement in speed of about 5% – 10%. However, the full impact of these measures has to be determined by detailed performance analysis with real world examples at a later point.

	a_1	a_2	a_3	a_4
b_1	*8*	*7*	*6*	*5*
b_2	*7*	*6*	*5*	*4*
b_3	*6*	*5*	*4*	*3*
b_4	*5*	*4*	*3*	*2*
b_5	*4*	*3*	*2*	*1*

Figure 2: Graphical representation of the multiplication algorithm for two numbers of length 4 and 5 respectively. Every cell in the grid represents one product. The coloring of the cells shows products of the same order of magnitude, assuming that the input numbers where well conditioned. The numbering shows the order in which these products are evaluated, so that we take advantage of that ordering.

Another way to reduce execution time is to "pre-split" the limbs of each number. In the worst case, every limb is multiplied by all limbs of the other number using a Dekker multiplication. If we split the numbers anew every time, we recalculate the same splitting over and over. Instead, the limbs are split once in the beginning and then stored in a local array. Whenever a multiplication takes place, it is performed using the pre-split limbs.

The last important issue with this algorithm is the order in which the limbs are multiplied. Clearly, the order of the multiplications is irrelevant, as long as it is ensured that every pairs is multiplied exactly once. The naive way would be to simply go straight through the first sequence a_i and multiply each limb with each b_j. However, if we want the input to the accumulator to be roughly ordered, it is better to multiply numbers that yield the same order of magnitude, starting with the lowest order (see Figure 2). This is a little more involved, but not much more computationally expensive than the naive approach.

Note last that, if the input numbers were normalized, it would be sufficient to just multiply the first limbs and immediately add the products of higher limbs to the error term. Since we cannot assume that input is normalized, however, this method cannot be applied to our algorithm. But since we do compare each floating point product to the error term, our algorithm does not perform significantly slower.

2.4.1 Squaring

Analyzing the algorithm for multiplication given above, one finds that some steps in the process are redundant if both arguments are the same, i.e. if a number is squared. For one, the splitting only has to be performed for one number. Also, we can utilize the fact that a multiplication by 2 is always correct in floating point arithmetics[1].

[1] Except for overflows, which we deal with separately.

	a_1	a_2	a_3	a_4
a_1	*10*	*9*	*8*	*7*
a_2	*9*	*6*	*5*	*4*
a_3	*8*	*5*	*3*	*2*
a_4	*7*	*4*	*2*	*1*

Figure 3: Graphical representation of the squaring algorithm for a number of length 4. Every cell in the grid represents one product. However, the off-diagonal products appear twice, so we can ignore half of them as shown in the illustration. The numbering shows the order in which these products are evaluated, so that we take advantage of well conditioned input.

So, to calculate

$$\left(\sum_i a_i\right)^2 = \sum_{i,j} a_i a_j = \sum_i a_i^2 + \sum_{i \neq j} a_i a_j = \sum_i a_i^2 + 2\sum_{j>i} a_i a_j$$

one simply goes through all indices i and, in an inner loop, lets j loop from $i+1$ to n. After the Dekker multiplication of the cross terms, one simply multiplies each part of the result by 2.

The algorithm is shown in Figure 3. Note that, as in the regular multiplication, to keep the input to the accumulator at least roughly ordered, we evaluate products of the last limb first, and work our way up to the larger terms. While this only results in approximate ordering up to the 6th product, the speed gain by cutting the number of Dekker multiplications in half clearly outweighs the losses in the accumulator.

2.5 Division

Division is different from the other algorithms presented so far. Instead of directly calculating the result using low level floating point operations of the processor, we use a constructive approach that produces more and more valid digits of the result.

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

The idea is to first approximate the high precision intervals by floating point numbers, and then divide those using floating point arithmetics. That number will be the first limb in the result. Then we use high precision operations to calculate the difference between our floating point result times the denominator, and the numerator, i.e. the error left over. Now

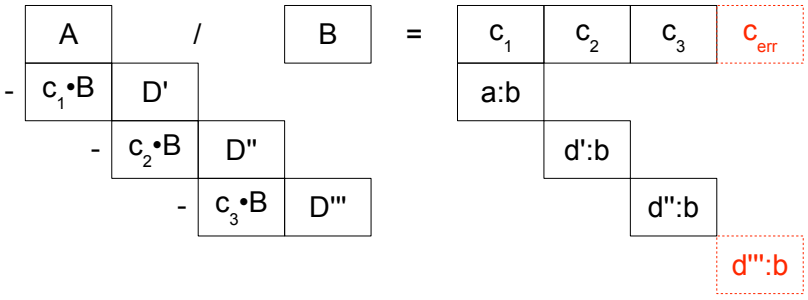

Figure 4: Graphical representation of the division algorithm for two numbers A and B. Capital letters represent high precision intervals, while the lower case letters are floating point numbers that represent the floating point sum of the limbs of the corresponding high precision interval. All arithmetic operations are Dekker operations, except for the floating point division, which is denoted by a colon (:).

we continue this process with this difference to get the next limb and so on. Once we have reach a high enough accuracy, or the maximal number of limbs, we can terminate the process and immediately have a rigorous error bound at hand.

In mathematical terms, we want to evaluate $C = A/B$. Then we can reformulate the problem like this:

$$C = A/B = c + D' \tag{9}$$

where $c = \sum a_i / \sum b_i$ evaluated in floating point arithmetics, and D' is a high precision interval that represents the error. This error is, barring the possibility of cancellation during the summation of the high precision intervals, of order of $\varepsilon \cdot c$. Solving Equation 9 for D' we get:

$$A/B = c + D' \;\Rightarrow\; A = Bc + D'B \;\Rightarrow\; \frac{A - Bc}{B} = D' \tag{10}$$

Let $D = A - Bc$. As noted before, D will usually be an order of ε smaller than A. To find the next limbs of the result, we have to determine D'. This, again, involves a division of the two high precision intervals D and B. The first limb of this can again be determined as described above, so one simply start over with the same algorithm, replacing A with D. This way, one recursively gets more and more limbs of the final result, while always keeping a rigorous error bound, namely D. This algorithm is graphically shown in Figure 4.

The error D allows, of course, immediately the calculation of an error bound for the final result. When the algorithm has determined enough limbs of the result, one can obtain a floating point interval enclosure of the error D. The same can be done for the divisor B. Then, using straightforward floating point interval division, it is possible to obtain an upper bound on the error term for the result.

Of course, this algorithm needs to test the denominator for a division by zero. In our implementation, this is done by calculating a floating point interval approximation of the divisor B, which is then tested to not contain zero. If the divisor does contain zero, an error flag is set for the result, marking it invalid.

2.6 Comparisons and Bounding

The following section describes algorithms for bounding and comparisons of high precision intervals. For the bounding operations we obtain an upper and lower bound as a double precision number. The comparison operations allow rigorous comparisons between two high precision intervals, or a high precision interval and a double precision number. Due to the interval nature of the high precision intervals, the meaning of the comparison operation also has to be specified.

2.6.1 Bounding

The bounding operation essentially converts a high precision interval into a double precision interval rigorously containing the high precision interval. The upper and lower bound of the result are, of course, upper and lower bounds for the high precision number.

To calculate such an interval enclosure the unevaluated sum $A = \sum_i a_i \pm a_{err}$ is evaluated in double precision interval arithmetic. The initial interval is just the error bound $[-a_{err}, a_{err}]$. Then each limb is added to this interval. To avoid excessive and unnecessary overestimation, the summation is carried out backwards, beginning with the last limb.

The resulting interval represents the desired enclosure for this operation.

2.6.2 Comparisons

Although high precision intervals are intervals, it is possible to define certain useful comparison operators to act on them. When comparing two real numbers, there are exactly three choices: one of them is either larger, smaller, or equal to the other. However, due to the interval nature of high precision intervals, these concepts have to be redefined. Note that in the following definitions, one of the intervals can be assumed to have zero width, yielding the condition for comparisons to real numbers.

Definition 5. *We say an interval A is larger than an interval B if the same is true for each pair of elements in the intervals:*

$$A > B \Leftrightarrow x > y \mid \forall x \in A \; \forall y \in B$$

Definition 6. *Similarly, we say an interval A is smaller than an interval B if the same is true for each pair of elements in the intervals:*

$$A < B \Leftrightarrow x < y \mid \forall x \in A \; \forall y \in B$$

Equality, and subsequently inequality, is somewhat harder to define. Of course, one could use the mathematical definition of equality of two intervals

$$A = B \Leftrightarrow A \subseteq B \; \wedge \; B \subseteq A$$

However, for practical purposes this is not useful. After lengthy calculations, the chances that two intervals represented by a high precision intervals are exactly equal, are practically zero. This is particularly true if a program intends to check for a certain condition, such as

$A = 0$. In practical situations, this is never be true since even if A had only one limb of value 0, the error bound surely would not be zero.

Instead, we use the following definition for equality.

Definition 7 (Equality)**.** *We say A is equal to B, iff there is a non-zero intersection between the two intervals:*

$$A = B \Leftrightarrow A \cap B \neq \emptyset$$

Inequality, on the other hand, is defined as the negation of equality.

Definition 8 (Inequality)**.** *We say two high precision intervals A and B are not equal iff their intersection is the empty set:*

$$A \neq B \Leftrightarrow A \cap B = \emptyset$$

Note that, while being useful, these definitions do not have all mathematical properties of the same relations on real numbers. Clearly, equality in the above sense is not transitive, as is illustrated by the intervals $[0, 2] = [1, 4] = [3, 5]$ but $[0, 2] \neq [3, 5]$.

This definition of equality has mainly practical value. In typical interval code, it is often necessary to test if two intervals overlap, or if a certain value is included in an interval. With the above definition of equality, these kinds of tests can be carried out easily and somewhat elegantly in a program.

Implementation of comparisons With those definitions in place, we will proceed to discuss the implementation of these test. The above definitions give a binary result, that is either "true" or "false", for each comparison. However, in our implementation, those comparison operators will return a tertiary value. Additionally to "true" and "false" there is also a value representing an "undetermined" result. This value is returned, if the rigorous result of the comparison cannot be determined.

Comparing a High Precision Interval to a Floating Point Number To compare a high precision interval A to some floating point number b, we have to understand what this means in terms of the representation of A. Let $\lceil X \rceil = \sum_i x_i + x_{err}$ be the upper bound of the interval represented by high precision interval X, and $\lfloor X \rfloor = \sum_i x_i - x_{err}$ the lower bound. Then we can write the different conditions defined above as:

- $A > b$:
$$\lfloor A \rfloor > b \;\Rightarrow\; \sum_i a_i - a_{err} > b \;\Rightarrow\; \sum_i a_i - a_{err} - b > 0$$
- $A < b$:
$$\lceil A \rceil < b \;\Rightarrow\; \sum_i a_i + a_{err} < b \;\Rightarrow\; \sum_i a_i + a_{err} - b < 0$$

- $A = b$:

$$\begin{aligned} \lceil A \rceil \geqslant b \quad &\wedge \quad \lfloor A \rfloor \leqslant b \\ \Rightarrow \sum_i a_i + a_{err} \geqslant b \quad &\wedge \quad \sum_i a_i - a_{err} \leqslant b \\ \Rightarrow \sum_i a_i + a_{err} - b \geqslant 0 \quad &\wedge \quad \sum_i a_i - a_{err} - b \leqslant 0 \end{aligned}$$

These comparisons are implemented in a two step process. The first test simply calculates a double precision interval bound for each sum and compares the resulting interval to 0. For the test $A > b$, for example, the sum

$$I = \sum_i a_i - a_{err} - b$$

is evaluated in interval arithmetic. If the lower bound of the resulting interval I is larger than 0, the test succeeds and A indeed is larger than b as defined above. Similar calculations are done for the tests $A = b$ and $A < b$.

If the above test does not determine the result, i.e. if the interval I contains 0, this simple test cannot be used to determine the result of a comparison. Instead, a more sophisticated algorithm has to be employed.

From the above formulation it is obvious, that each comparison operation can be reduced to the case of determining the sign of a series of floating point numbers. Thus an algorithm is required, that can rigorously determine the sign of a series of floating point numbers using only floating point operations. Such an algorithm is described in subsubsection 2.6.3. It is used to determine the sign of each series.

If that algorithm is unable to determine the sign rigorously, the comparison cannot be carried out. The result returned is "undefined".

Comparing two High Precision Intervals Comparing two high precision intervals A and B is a little bit more involved. Nonetheless, this can also be reduced to the determination of the sign of a series of floating point numbers. Unfortunately, one cannot simply take the difference $C = A - B$ and compare that to 0, because the subtraction will only yield an interval enclosure of the result. The result C, however, may be wider than it has to be due to overestimation. Thus it is possible that, if two disjoint intervals A and B are very close to each other, the subtraction yields in interval that includes 0. This leads to an incorrectly detected intersection of the two intervals.

In terms of the representation of A and B, and using the same notation as before, the comparison of the two high precision intervals can be written as:

- $A > B$:

$$\lfloor A \rfloor > \lceil B \rceil \;\Rightarrow\; \sum_i a_i - a_{err} > \sum_j b_j + b_{err} \;\Rightarrow\; \sum_i a_i - a_{err} - \sum_j b_j - b_{err} > 0$$

- $A < B$:

$$\lceil A \rceil < \lfloor B \rfloor \;\Rightarrow\; \sum_i a_i + a_{err} < \sum_j b_j - b_{err} \;\Rightarrow\; \sum_i a_i + a_{err} - \sum_j b_j + b_{err} < 0$$

- $A = B$:

$$\begin{aligned} & \lceil A \rceil \geqslant \lfloor B \rfloor \quad \wedge \quad \lfloor A \rfloor \leqslant \lceil B \rceil \\ \Rightarrow\; & \sum_i a_i + a_{err} \geqslant \sum_j b_j - b_{err} \quad \wedge \quad \sum_i a_i - a_{err} \leqslant \sum_j b_j + b_{err} \\ \Rightarrow\; & \sum_i a_i + a_{err} - \sum_j b_j + b_{err} \geqslant 0 \quad \wedge \quad \sum_i a_i - a_{err} - \sum_j b_j - b_{err} \leqslant 0 \end{aligned}$$

As in the case for the comparison to a single double precision number, an interval bound for each of the series is calculated. If that approximate interval bound is sufficient to determine the result of the comparison, the correct result is returned.

Otherwise the sign of the sum is determined using the algorithm described in the next section. If that algorithm is unable to determine the sign rigorously, the comparison cannot be carried out. The result returned is "undefined".

2.6.3 Sign of a Floating Point Series

Now we have reduced comparisons of floating point numbers to the determination of the sign of a series of floating point numbers. To answer this question exactly, and only using floating point arithmetics, we have developed the following algorithm based on a simple bounding argument. Let the series be given by n floating point numbers a_i. The goal is to rigorously determine sign of the sum $\sum_{i=1}^{n} a_i$.

Without loss of generality, let $|a_1|$ and $|a_2|$ be the largest and second largest absolute values of all n numbers. Then:

$$\sum_{i=1}^{n} a_i = a_1 + \sum_{i=2}^{n} a_i \leqslant a_1 + |\sum_{i=2}^{n} a_i| \leqslant a_1 + \sum_{i=2}^{n} |a_i| \leqslant a_1 + (n-1) \cdot |a_2| \tag{11}$$

$$\sum_{i=1}^{n} a_i = a_1 + \sum_{i=2}^{n} a_i \geqslant a_1 - |\sum_{i=2}^{n} a_i| \geqslant a_1 - \sum_{i=2}^{n} |a_i| \geqslant a_1 - (n-1) \cdot |a_2| \tag{12}$$

Let n^* be an integer power of two, that is greater than or equal to $n-1$. Then the inequalities obviously still hold, if $n-1$ is replaced by n^*. Furthermore, all operations needed to calculate $n^* \cdot |a_2|$ are exact in floating point arithmetic. Combined with the fact that comparisons of floating point numbers are always exact, we can therefore conclude that the sign of the whole sum will be the same as the sign of a_1 if

$$|a_1| > n^* \cdot |a_2| \tag{13}$$

If, however, a_1 and a_2 are too close together, we can perform a Dekker addition of the two floating point numbers. That is guaranteed to result in two different floating point numbers,

which rigorously replace a_1 and a_2 in the above sums without changing their value. Also they fulfill $|a_1| > 1/\varepsilon \cdot |a_2|$ and, if n^* in the above equation is smaller than $1/\varepsilon$, also $|a_1| > n^* \cdot |a_2|$. Repeating the process with this new series of numbers will provide different values for a_1 and a_2, and thus may allow to determine the sign.

Unfortunately, at this point we cannot guarantee that this algorithm does terminate and always determines the sign of a sequence. In our implementation, the maximal number of iterations performed is limited. If after that predefined number of iterations the sign could not be determined, the function returns an error, and the calling functions act accordingly.

There are a few special cases to be considered. In the trivial case of $n = 1$ then the sign of the sum is clear. If the largest value turns out to be $a_1 = 0$ then the sign is also clear, since then the whole sum has to be equal to 0. As in our other algorithms, we do not consider over- and underflow of floating point numbers at this point.

2.6.4 Absolute Value

The interval nature of high precision intervals also requires a new definition of the absolute value. For this operation, the high precision implementation follows the currently existing implementation of double precision intervals in COSY INFINITY. It returns a single double precision number that is guaranteed to be an upper bound on the absolute values of every element in the interval.

Definition 9 (Absolute Value)**.** *The absolute value of a high precision interval A returns one double precision number v such that*

$$v \geqslant |x| \ \forall \ x \in A$$

Note that there is no requirement for v to be in any way optimal. Any v that satisfies the condition in Definition 9 is acceptable. To make this intrinsic useful, however, it is clearly in the interest of the implementation to avoid large overestimation.

In our implementation the upper bound v is calculated by first obtaining a double precision interval enclosure of the high precision number A (see subsubsection 2.6.1). Then v is set to be the larger of the absolute values of the upper and lower bounds of the double precision interval. Since the absolute value is an exact operation in floating point arithmetic, this result does satisfy the definition of the absolute value for high precision intervals.

2.7 Intrinsic Functions

Intrinsic functions of a high precision interval X are implemented using either an iterative Newton method or by the Taylor expansion of the intrinsic function.

A Newton method is used for the square root. The advantage here is, that Newton iterations converge very quickly, and we can utilize the underlying floating point operations to easily obtain a good starting value. Argument reduction, as with the Taylor series, is usually not necessary. Also, for the root finding functions the Newton iteration involves only relatively cheap, basic high precision operations.5

To evaluate the exponential, logarithm, or trigonometric functions, we rely on the Taylor expansion of the respective function. Here, we successively calculate each order, and add

it to the result. By keeping the previous order term after adding it to the result, the cost of calculating the next order remains constant, so the computational cost of evaluating the series is linear in the order of the expansion.

To decide when to terminate the expansion, we use the current order term. Since in any useful Taylor series the terms have to be monotonously decreasing, once the next order term becomes smaller than the error bound of the result, the expansion can be terminated. All further terms at that point would only contribute to the error term of the result.

To speed up convergence in the Taylor series, it is also important to reduce the arguments so that they are small enough to yield fast convergence. This is achieved by exploiting symmetries in the functions involved.

To rigorously estimate the error bound of the result, we use the Taylor remainder formula Equation 7 where applicable. If that is not possible, we have to revert to other methods geared specifically towards the given expansion.

2.7.1 Square Root

To compute the square root X of a high precision number A, we use the interval Newton method [11] to obtain an interval enclosure of the solution of

$$X^2 - A = 0.$$

This yields the iteration step

$$X_{n+1} = M(X_n) - \frac{M(X_n)^2 - A}{2X_n} \tag{14}$$

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

Provided that the initial interval enclosure X_0 contains the square root, each X_n contains the correct result. Furthermore, the sequence of intervals converges quickly, even without intersecting each X_n and X_{n+1}. The criterion for terminating this iteration is the width, i.e. the error bound, of X_n. Once the intervals do not contract any more, we have reached the best possible enclosure of the result.

The choice of the initial starting value X_0 is done in double precision interval arithmetic. First, an interval enclosure I of the high precision interval A is calculated (see subsubsection 2.6.1). If this enclosure contains values less than or equal to 0, the resulting high precision interval is set to 0 and the error flag for that number is set. Otherwise, the interval square root of I is converted into a high precision number, which serves as the initial interval X_0.

Provided that the double precision interval operations are rigorous, the initial interval X_0 does contain the exact square root, and thus fulfills the requirements for the interval Newton algorithm.

Note that in our representation of high precision intervals it is particularly easy to obtain an the midpoint of an interval exactly. 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 X_{n+1} really contains the correct interval.

2.7.2 Inverse Square Root

While it would be possible to calculate the inverse square root using another interval Newton method, we simply calculate the square root and then invert the result. This is slightly less efficient, as computation requires one more division, but the loss in performance is almost insignificant.

2.7.3 Exponential and Logarithm

Exponential To calculate the exponential of a high precision number X, we simply use the Taylor expansion of the exponential function [4]:

$$\exp X = \sum_{n=0}^{\infty} \frac{X^n}{n!} = 1 + X + \frac{X^2}{2} + \frac{X^3}{6} + \dots \tag{15}$$

Since the exponential function for large arguments obviously grows very fast, it is necessary to reduce the argument to a small value to ensure convergence in reasonable time. To achieve that, we make use of the identity

$$\exp(x \cdot 2^k) = \exp(x)^{(2^k)} \tag{16}$$

First notice that $\exp(700) > 10^{304}$, which is near the largest representable double precision floating point number. Similarly, of course, $\exp(-700) < 10^{-304}$ which again is near the smallest representable double precision floating point number. We thus limit the domain on which the exponential function is computed to $D = [-700, +700]$. If a value is larger than 700, it will produce an error, while a value smaller than 700 will always return a predetermined interval $[0, d]$ where $d > 10^{-304}$.

Given some value $x \in D$, we define $y = x \cdot 2^{-11}$. This ensures that $|y| \leqslant 700/2048 < \frac{1}{2}$. By Equation 16 we then have

$$\exp(x) = \exp(y \cdot 2^{11}) = \exp(y)^{(2^{11})}.$$

Because of the choice of the factor 2^{-11}, y is of sufficiently small size such that the Taylor series for $\exp(y)$ converges relatively quickly. As a criterion for terminating the expansion, we use the size of the nth computed coefficient of the expansion. If that coefficient is less than the error bound of the result, all further coefficients will merely increase the error bound, and thus the expansion can be truncated at that point.

To estimate the remainder for that series, we use that

$$\left(\frac{d}{dx}\right)^n \exp(y) = \exp(y)$$

and

$$\sup(\exp([-0.5, 0.5]) = \exp(0.5) = \sqrt{e} \leqslant 2$$

Thus, the Taylor remainder term of the exponential series computed up to order n can be bounded above by twice the next term in the expansion:

$$|R_{n+1}| \leqslant 2 \cdot \left| \frac{X^{(n+1)}}{(n+1)!} \right| \tag{17}$$

This yields a rigorous enclosure of the value $\exp(y)$. In order to obtain the final result for $\exp(x)$, the result of the series evaluation has to be raised to the power of $2^1 1 = 2048$. Fortunately, this can be done very fast since for any x

$$x^{(2^n)} = \left(x^{\left(2^{n-1}\right)}\right)^2 = \left(\left(x^{\left(2^{n-2}\right)}\right)^2\right)^2 = \cdots = \left(\left(\ldots\left(x^2\right)^2\ldots\right)^2\right)^2$$

To evaluate $\exp(y)^{(2^{11})}$ one simply squares $\exp(y)$ repeatedly 11 times. Since squaring is the second cheapest operation in our arithmetic, these operations incur no performance penalty.

Natural Logarithm To evaluate the natural logarithm of a high precision number X, we utilize the following expansion based on the hyperbolic areatangens function [4]:

$$\ln X = \sum_{n=0}^{\infty} \frac{2}{2n+1} \left(\frac{X-1}{X+1}\right)^{2n+1} = 2\frac{X-1}{X+1} + \frac{2}{3}\left(\frac{X-1}{X+1}\right)^3 + \frac{2}{5}\left(\frac{X-1}{X+1}\right)^5 + \ldots \tag{18}$$

Obviously, this series converges the faster the closer X is to 1. Also, since X has to be positive for the logarithm to be defined, the argument for the series is always less than or equal to 1, and therefore the sequence is monotonously decreasing. That allows us to use the current order as a termination criterion for the evaluation of the series.

The series given above converges very slowly for X close to 0, or large values of X. We hence reduce the argument to be close to 1 using the identity:

$$\ln(x) = m\ln(2) + \ln(2^{-m}x) \tag{19}$$

Choosing m to be the nearest integer to the 2 based logarithm, i.e.

$$m = \lfloor \ln(x)/\ln(2) + 0.5 \rfloor,$$

the new argument satisfies the inequality $|2^{-m}x - 1| \leqslant \frac{1}{3}$. For those arguments, the Taylor series converges relatively quickly. Note that even if m is not computed exactly, all operations remain verified, they are just not optimally fast any more.

To estimate the remainder of the natural logarithm, we use the following remainder to the first n terms in the series given in Equation 18:

$$|R_{n+1}| \leqslant \left| \frac{(X-1)^2}{2X} \left(\frac{X-1}{X+1}\right)^{2n} \right| \tag{20}$$

2.7.4 Trigonometric Functions

Sine We use the well known expansion of the sine function [4]:

$$\sin X = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)!} X^{2n+1} = X - \frac{X^3}{3!} + \frac{X^5}{5!} - \dots \tag{21}$$

$$R_n = \frac{X^{n+1}}{(n+1)!} \tag{22}$$

In the remainder term we used that $\left(\frac{d}{dx}\right)^{n+1} \sin(x) \leqslant 1$. To reduce the argument, we take two steps. First, we exploit the periodicity of the sine function, $\sin(x) = \sin(x \mod 2\pi)$, which allows us to reduce the argument to lie within the interval $[-2\pi, 2\pi]$. If the absolute value of the result of that reduction is still larger than π, we use the symmetry of the sine function, $\sin(x) = -\sin(x \pm \pi)$. Depending on the sign of the argument we either add or subtract π. This reduces the argument into the interval $[-\pi, \pi]$.

While further reduction is possible, for example based on the half angle formula, these two steps seem sufficient to provide fast convergence in the series. In all calculations we use a pre-calculated, rigorous value of π, that is stored as a global high precision variable within COSY.

Cosine We use the well known expansion of the cosine function [4]:

$$\cos X = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} X^{2n} = 1 - \frac{X^2}{2!} + \frac{X^4}{4!} - \dots \tag{23}$$

$$R_n = \frac{X^{n+1}}{(n+1)!} \tag{24}$$

This is almost identical to the sine function. For the remainder bound, we again use that $\left(\frac{d}{dx}\right)^{n+1} \cos(x) \leqslant 1$ applied to the Taylor remainder formula. Argument reduction is performed exactly as described for the sine function. Clearly, all properties used in the argument reduction for the sine function are also valid for the cosine function.

Tangent The Taylor expansion of the tangent is not as trivial as that of the sine and cosine. Instead of calculating the tangent directly by its Taylor series, we rely on the definition of the tangent:

$$\tan(X) = \frac{\sin(X)}{\cos(X)} \tag{25}$$

While this requires two full evaluations of the sine and the cosine, the performance impact can be reduced by performing the argument reduction only once for both the sine and cosine evaluation.

This is done based on the identity $\tan(x) = \tan(x \mod \pi)$. This allows us to reduce the argument to be within the interval $[-\pi, \pi]$. All error bounding of the tangent relies on the correct error bound of the sine and cosine functions.

Arc Sine The arcsine function relies on the arctangent, as given by the identity [4]:

$$\arcsin X = 2 \arctan \frac{X}{1 + \sqrt{1 - X^2}} \tag{26}$$

Arc Cosine The arccosine function relies on the arctangent, as given by the identity [4]:

$$\arccos X = \frac{\pi}{2} - \arcsin X \tag{27}$$

Note that in the actual implementation a pre-calculated, rigorous interval enclosure is used for π .

Arc Tangent For the arctangent we use the following expansion [4]:

$$\arctan X = \sum_{n=0}^{\infty} (-1)^n \frac{X^{2n+1}}{2n+1} = X - \frac{1}{3}X^3 + \frac{1}{5}X^5 + \dots \tag{28}$$

This series converges for $|X| \leqslant 1$. To reduce the argument to that range, we make use of the following identity [4]:

$$\arctan(X) = 2 \arctan \frac{X}{1 + \sqrt{1 + X^2}} \tag{29}$$

Applying this equation once, reduces all arguments from $(-\infty, +\infty)$ to $(-1, 1)$. Because convergence of the series for that range still is slow, applying it a second time ensures the argument to lie within $(-\frac{1}{1+\sqrt{2}}, \frac{1}{1+\sqrt{2}})$. After evaluating the series for the reduced argument, only a multiplication by 4 is necessary. This is, even in floating pint arithmetic, an exact operation.

Note that this reduction could be repeated a third time, reducing the argument to $|x| \leqslant 0.25$. However, it is questionable if that actually increases the speed of convergence enough to warrant the additional computation needed for the reduction.

To get the error bound, it is not possible to use the Taylor remainder term. Careful examination of the term $\sup \left(\frac{d}{dx}\right)^n \arctan x$ 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. Instead, we use the fact that this series has monotonously decreasing elements and alternating signs. Thus the Leibniz criterion (subsection 1.7) applies, and an estimate of the remainder of the partial series is given by:

$$R_n = \left| \frac{X^{2n+3}}{2n+3} \right| \tag{30}$$

2.7.5 Hyperbolic Functions

For these functions we rely on the expression of the hyperbolic functions in terms of the real exponential function [4]. The overhead due to this is minimal, since each hyperbolic function only requires one evaluation of the exponential function.

Hyperbolic Sine Transforming the well known form of the hyperbolic sine into an equal, yet computationally superior form, we obtain the relation

$$\sinh X = \frac{\exp(X) - \exp(-X)}{2} = \frac{\exp(X)^2 - 1}{2\exp(X)} \tag{31}$$

This equation is then evaluated directly in high precision interval arithmetic.

Hyperbolic Cosine Analogously to the hyperbolic sine we obtain the relation

$$\cosh X = \frac{\exp(X) + \exp(-X)}{2} = \frac{\exp(X)^2 + 1}{2\exp(X)} \tag{32}$$

This equation is then evaluated directly in high precision interval arithmetic.

Hyperbolic Tangent Similarly, for the hyperbolic tangent we derive the relation

$$\tanh X = \frac{\exp(X) - \exp(-X)}{\exp(X) + \exp(-X)} = \frac{\exp(X)^2 - 1}{\exp(X)^2 + 1} \tag{33}$$

As in the preceding cases, this equation is evaluated directly in high precision interval arithmetic.

2.8 Input / Output

Rigorously printing floating point numbers, and reading input from a string is essential to verified calculations. The functions provided by the underlying programing language are usually insufficient for this task. This is because the output will be converted to decimal fractions, which leads to truncation and thus incorrect results. The input functions provided are implemented to read input into double precision numbers. However, they are non-rigorous, and typically not even correct rounding is guaranteed.

Therefore, it is necessary to implement specific functions to perform these tasks. Although it might be possible to use some conversion functions provided by the system libraries, we will not rely on any of those. Instead, we implement all functions on our own. This not only guarantees portability, it also allows to ensure that every step of the conversion procedure is performed rigorously.

2.8.1 Rigorous Input

There are two types on input: machine readable input, that is typically generated by an earlier run of a computer program, serves the purpose to restore certain data structures from persistent storage into the programs memory. This type of input is not intended to be read or manipulated by humans. It is stored in a format suitably close to the internal representation of numbers (see last paragraph of subsubsection 2.8.2). If implemented properly, this input can be read into program memory without loss of precision.

The more challenging type of input are human readable numbers. In computer programming languages it is common to write decimal numbers in what is known as *scientific notation*. A conversion algorithm has to perform several tasks:

1. Parse the input string, and make sure it conforms to the specified format
2. Verify that the number is within the limitations of the high precision numbers (see subsubsection 2.1.1)
3. Rigorously convert the number from the decimal system into the representation used internally

First, it is necessary to define the input format. Our input routines accept a format that is based on the scientific notation. More specifically, numbers consist of several fields, some of which are optional. They are separated by specific character constants.

$$[+-]^?[0123456789]^*[.,]^?([0123456789]^*)[eEhH]^?([+-]^?[0123456789]^*) \tag{34}$$

In this representation, which is loosely based on POSIX regular expressions, characters in square brackets represent the allowed character set for the respective field. Any character listed in the square brackets is an allowed character. The superscript behind the square brackets designates the number of characters from the preceding set, that are allowed in the field. $?$ means exactly none or one character appear. $*$ means zero or more characters appear. Expressions in parentheses are only allowed to appear, if the field directly before is not empty.

Expression 34 dictates, that each number has an optional sign of either $+$ or $-$. This is followed by zero or more digits representing the integer part of the number represented in base 10. This then is followed by an optional decimal point, which, to accommodate users from Europe as well as from the U.S., can be either a decimal dot . or a comma ,. Only if this decimal separator is present, it is followed by an optional fractional part in base 10. The mantissa is then followed by an optional exponent separator which can be one of e, E, h or H. If this separator is present, it can be followed by another optional sign and a base 10 integer of zero or more digits.

To illustrate this format, here are some examples of valid and invalid numbers:

- $0505,1983E24$ is a valid number
- $.666$ is a valid number
- $1,0000000000000000000$ is a valid number
- 17 is a valid number
- $-0.000911000E$ is a valid number (the missing exponent will be interpreted as 0)
- $-1E-99$ is a valid number
- $.E17$ is a valid number (it will be read as zero since the empty mantissa will be interpreted as 0.0)
- $-.$ is a valid number (it will again be read as zero)[2]

[2] Although this and the preceding examples are valid, it is not recommended to use this representation since it is confusing. Use 0 instead.

- $5;1983E17$ is an invalid number (; is not a decimal delimiter)
- . 8888 is an invalid number (illegal spaces after the decimal delimiter)
- $1,000.0$ is an invalid number (two decimal delimiters)
- 17 is an invalid number (illegal spaces in front)
- $1*10**22$ is an invalid number (illegal character $*$)
- $1A7EFF$ is an invalid number (illegal characters A and F, all input has to be base 10)

In addition to this format, the conversion functions also accept certain special strings, which are handled as completely separate cases. Currently, those character constants are (case-insensitive)

1. "PI" - The value of the pre-calculated interval representing π in the current precision
2. "2PI", "2*PI" - The value of the pre-calculated interval representing 2π in the current precision
3. "PI/2" - The value of the pre-calculated interval representing $\pi/2$ in the current precision

This allows for certain constants to be obtained easily, without requiring the user to calculate them in their program.

Another important issue is the accuracy of the resulting numbers. Some numbers that have a finite fractional representation in decimal notation, do not have such a representation in binary. Numbers as simple as 0.1 cannot be represented by a finite binary fraction. So, when converting, we have to choose a precision up to which we want our numbers to be accurate. In our implementations we obviously use the default precision supplied by the user at runtime by the `HIINI` routine (see subsubsection 3.1.2).

The conversion algorithm attempts to encode at least the current number of digits exactly, possibly more. In any case, it always attaches an error bound to result, so that the number represented by the string is guaranteed to be contained in the resulting high precision interval.

2.8.2 Rigorous Output

When printing rigorous numbers, it is necessary to avoid conversion from the floating point number system used in the computer representation of a number. We print double precision numbers rigorously, by outputting their floating point representation exactly.

In order to extract the relevant parts of the bit pattern of a double precision number, we rely on IEEE 754 [2] conform storage. This means, that the number is represented internally as

$$s \cdot m \cdot 2^e \tag{35}$$

where the numbers are integers of different size: s is the sign (1 bit), m is the mantissa (52 bits[3]) and e is the exponent (11 bits). This results in a total of 64 bits.

To achieve rigorous output we extract all of these fields form the floating point number a, and write it as

$$\pm m_a b e_a$$

where m_a is the mantissa and e_a is the exponent. The character "b" as a separator is reminiscent of the "e" in scientific notation. Thus the decimal number 4.25 is output as $17b-2$ meaning $17 \cdot 2^{-2}$.

When outputting a high precision interval, all limbs are printed once in this way, and once as non-rigorous, decimal fractions. This way, the output is human and machine readable at the same time. If desired, the machine readable output can then be used to obtain a decimal fraction for the high precision interval using third party packages.

3 Implementation

In this section, we present some implementation specific details. We realized the algorithms described above in FORTRAN. They are integrated into the rigorous computation package COSY INFINITY [9], where the high precision intervals are available as the HI data type.

Throughout the implementation, we follow the naming convention of the existing COSY code. Each elementary operation is abbreviated by one letter – `A` for addition, `S` for subtraction, `M` for multiplication, and `D` for division. The type of the arguments is pre- / appended to that letter. A high precision high precision interval is abbreviated by `DE`, while a single floating point number uses `RE`. Thus the division of a floating point number by a high precision interval is carried out by the function `REDDE`.

3.1 Data Storage

The data associated with a high precision interval is stored in the COSY main memory. As for regular COSY variables, the memory index of the error term of a HI variable is stored in `NEND`. The maximum memory allocated for the COSY variable itself may be more than the that. Each HI number has to have at least one limb, even if it is zero. Thus, together with the error term, the shortest length for a COSY variable holding a HI is two. Based on the limitations of our representation, the maximum length of a HI interval in COSY is set to 15 limbs, or about 225 decimal digits.

The limbs and the error bound are stored in the array `CC`. Limbs are stored consecutively starting at the first memory location of the variable. After the limbs follows one more entry for the error bound. `NEND` always points to the error bound. In `NC`, error flags for the HI number are stored.

[3] Actually it is 53 bits because the first bit is implicitly assumed to be one.

3.1.1 The Error Flag

In the first entry in `NC()` for a variable, we store an additional error flag. Any value other than zero represents an error condition, i.e. the high precision interval represented by this number is not a valid interval. Currently, there is only one error code, 1, which represents any error. However, in future releases there may be different codes.

During calculations, every subroutine checks for those error conditions first. If any of the arguments are defective, the result is set to be defective as well, and the subroutine returns immediately without further calculations. The error code of the result is set to be the maximum of the error codes of the arguments. The idea here is that in future releases the error codes may signify the severity of an error condition. This way, always the most severe condition is propagated.

When printing a defective number, the output will read "DEFECTIVE NUMBER". The value of the interval itself is ignored and not printed.

3.1.2 Initialisation

The built-in procedure `DEINI` in COSY is used to initialize the high precision interval engine. As a parameter, this procedure is given the number of decimal digits required. This function must be called once in the beginning of every COSY program, before any high precision interval operations are performed. It can be called subsequently during the program flow to change the current precision. Numbers that were calculated before such a call keep their precision, however, all subsequent calculations will be performed in the newly set precision.

The number of digits requested is converted into the number of limbs needed to represent that precision. This is done under the assumption that every limb adds 15 decimal digits to a high precision interval. So, to calculate the number of limbs n necessary to represent d decimal digits, we use the equation

$$n = \lfloor d/15 \rfloor + 1$$

All subsequent operations are limited to use at most that many limbs to represent the result. If the variable holding the result does not have sufficient memory, the result is truncated to fit into the variable. Truncated limbs are rigorously added to the error term.

Also, in this function some constants needed frequently in subsequent operations, are pre-calculated. Currently, this only includes the values of π, $\pi/2$, and 2π, as well as $\ln 2$. Those values are stored in global COSY variables, which are allocated at startup time, after other COSY constants are allocated. Of course, the precision of these constants is always the same as requested by the parameter. These constants are recalculated at the new precision every time `DEINI` is called throughout the program.

3.2 Intrinsic Functions

In all intrinsic functions, we split arguments if their relative width is larger than 10^{-15}. Each boundary is then evaluated separately, and the result is assembled using properties such as monotonicity of the intrinsic functions. This way, overestimation due to blow-up and dependency is reduced greatly in the evaluation.

To facilitate this, we have split each intrinsic function evaluation in two parts. The first procedure determines if the argument needs to be split, performs the splitting, and evaluates each boundary separately, merging the result in the end. The second procedure actually evaluates the Taylor series for the argument, or performs the Newton iteration. For some intrinsics requiring complicated argument reduction, another procedure is called before the evaluation to reduce the argument.

Furthermore, for intrinsic functions with limited domain, the argument is verified to be within the domain by bounding it by a double precision interval. This may exclude certain very specific boundary cases from evaluating, but it does make those checks very fast and does not really limit the functionality. If the argument of an intrinsic is determined to be outside the allowed domain, the error flag of the result is set to 1, while the result itself is initialized as 0. The intrinsic then returns immediately without further calculation.

3.3 Input / Output

In these algorithms, we make implicit assumptions about the machine representation of floating point numbers. Consequently, these parts will be dependent on the platform and machine architecture the code runs on. Our implementation is tested on x86 based machines, that use IEEE 745 double precision numbers.

3.3.1 Rigorous Input

To implement an algorithm that fulfills the requirements laid out it subsubsection 2.8.1, we implement two separate functions. The first function will act as a wrapper between COSY and the actual conversion. In this function, the COSY string is converted into a FORTRAN string, and the special strings are handled. Then the actual conversion function is called to parse the FORTRAN string into a high precision interval.

Parsing the string is done by stepping through the input character by character, and checking if it matches the allowed character set of the current field. If it does, we consume as many following characters as possible until a non-matching character is hit. At that point, the current field must have ended, and we continue with the next field description. Whenever a new field is entered, the beginning index is stored for later use. Similarly, the ending index is stored whenever a field has ended. After completing all fields, we thus have beginning and end indices for all fields.

If there are characters left at the end of the string, this means the input did not match our format. In that case an error is returned. Next, the exponent and its sign are extracted from the string. To correct the exponent, the number of digits found in the mantissa after the decimal point is subtracted. The new effective exponent is verified to lie within the limits described in subsubsection 2.1.1.

the next step is reading the mantissa. The mantissa of a regular double precision number has 53 bits. Thus, we can store numbers between 0 and $2^{53} \approx 9 \cdot 10^{15}$ exactly. Since our floating point model is round to nearest, if the result of an operation is representable as a floating point number, the operation is exact. Therefore, all operations on floating point

numbers are exact, as long as the numbers lie between 0 and $9 \cdot 10^{15}$. Consequently, any 15 digit integer in decimal representation can be exactly represented by a floating point number.

To convert the mantissa digit string, we loop through the mantissa from left to right, and assemble the digits into a `DOUBLE PRECISION` number. After 15 digits, the high precision interval representing the result is multiplied by 10^{15} rigorously. This operation shift all previously read digits 15 places to the left. Then the double precision number read from the mantissa string is added rigorously to the result.

This process is repeated until all digits in the mantissa are read. In the last step, care has to be taken to add to the result correctly. Instead of multiplying by 10^{15}, the correct factor is $10^{\#ofdigitsread}$, to ensure the correct shift. After this process, the result contains a rigorous representation of the integer mantissa.

The last step is to multiply or divide the result by $10^{|exp|}$. To evaluate $10^{|exp|}$ efficiently, we have devised the following algorithm. Since squaring numbers is cheap, it is beneficial to use that operation.

Expressing the exponent in binary form, $exp = b_0 b_1 b_2 b_3 \ldots$ with $b_i \in \{0, 1\}$, we obtain:

$$10^{exp} = 10^{(b_0 \cdot 2^0 + b_1 \cdot 2^1 + b_2 \cdot 2^2 + b_3 \cdot 2^3 + \ldots)} = \prod_i 10^{b_i \cdot 2^i}$$

Only the i with $b_i = 1$ contribute to the product since for $b_i = 0$ we have $10^{b_i \cdot 2^i} = 10^0 = 1$. Calculating those remaining factors is easy, considering:

$$10^{(2^i)} = \left(10^{(2^{i-1})}\right)^2 = \left(\left(10^{(2^{i-2})}\right)^2\right)^2 = \cdots = \left(\left(\ldots\left(10^2\right)^2\ldots\right)^2\right)^2$$

Let E be a high precision interval containing 1 initially. Starting with a high precision interval A containing $10^{(2^0)} = 10^1 = 10$, A is squared successively. If the corresponding bit is set in the exponent, then E is multiplied by A. Since the exponent is limited to be smaller than 512, there are at most 9 bits to check.

Multiplying the high precision mantissa by E, and adjusting the signs of the limbs according to the sign of the given string, completes the conversion into a high precision interval.

3.3.2 Rigorous Output

As described in subsubsection 2.8.2, rigorous human readable output will be achieved by outputting the integers that form a IEEE 754 double. Again, we have split the code into several functions for convenience.

The first function extracts the mantissa, exponent and sign from a double precision number. To extract this information from the `DOUBLE PRECISION` variable in FORTRAN, we use the bit field functions introduced into FORTRAN by DoD military standard 1753 [1] (also part of the Fortran 90 standard), and the concept of `EQUIVALENCE`, which allows different FORTRAN variables to share the same physical memory.

We declare a 64 bit integer variable using `INTEGER*8` as data type[4]. Then we declare this variable to be equivalent to a local `DOUBLE PRECISION` variable. Assigning the value

[4] This construct is not FORTRAN 77 standard, but it seems to be widely supported by all current compilers.

in question to the local double precision variable, we can now operate on the bit pattern of the double precision number. Using bit field functions, we extract the relevant bits from the double precision number.

According to IEE 745 [2], the bit layout of a double precision number is as follows. Bits are numbered from 0 through 63, starting with the right most bit. This is in accordance with the numbering scheme used by the FORTRAN bit field functions.

The layout of the memory of a double precision number is given by the following table:

63	62	...	52	51	...	40	...	30	...	20	...	10	...	0
sign	exponent			mantissa										

To store the mantissa, a 64 bit integer is required, since there are more than 32 bits. The exponent and the sign are both stored in a regular FORTRAN integer (32 bits). To extract the bits, the FORTRAN bit flied extensions provide the very useful function `IBITS(m, i, len)`. It extracts `len` bits from `m` starting at `i`, and returns a new integer that contains those bits flushed to the right, and the rest of it padded with zeros. This, of course, is exactly what we need to extract the parts:

```
EXPONENT = IBITS( TEMP, 52, 11 )
MANTISSA = IBITS( TEMP, 0, 52 )
SIGN = IBITS( TEMP, 63, 1 )
```

After extracting this information from the double precision number, we have another function interpret it and actually converting it into a character string. All strings returned are 27 characters long and right justified.

According to the IEEE754 standard, there are several cases:

MANTISSA = EXPONENT = 0 The number has the value 0.0. IEEE allows zeros to be signed, so the sign bit is still valid. This number is converted into $\pm 0b0$.

EXPONENT = 2047 signifies special cases, depending on the value of the mantissa:

MANTISSA = 0 The number is infinity. The sign bit is valid. This number is converted into the string $Infinity$.

MANTISSA $\neq$ 0 The number is not valid, i.e. not a number. This is converted into NaN ("not a number").

Other values For all other combinations, the correct value must be calculated according to these rules: The exponent is biased with a biasing constant of 1075, while in the mantissa the implicit leading bit, 52, is set to one. The converted string is then $\pm m\ b\ exponent$.

The reason for the biasing constant being different from the one given in [2] is that we shift the decimal point of the mantissa to the end, so it becomes an integer instead of a binary fraction. That way, we have to subtract an additional 52 from the exponent.

4 Performance

The performance of these algorithms is very important to us. Due to the integration into the High Precision Taylor Models, we are most concerned about the performance of high precision intervals with up to about 4 limbs. These will be the most common cases in Taylor Models.

In this section we briefly describe, how we plan to measure the performance of our algorithms. So far, no conclusive testing has been done yet.

4.1 Performance Monitoring

To better understand how our algorithms are doing, and if all the little optimizations we added are really worth the effort, we added some special debugging code. This code is only included in debug builds, where it counts the occurrences of certain events. We call this part of the code the *performance monitor* or PERFMON.

PERFMON is implemented as a named common block that contains one integer for each monitored event. In the source code, we simply add 1 to the respective counter, whenever that event is encountered. This is fairly non-intrusive as far as the performance of the algorithm is concerned, but still allows us to gain good insight into how well the different parts of our algorithms are performing in real longer real-world calculations.

Take, for example, the most important piece of code, the accumulator (**??**). There are several different things to monitor. First, of course, we are interested in how often the accumulator itself is called. Then we want to monitor how often the main loop is run. With this information, we can, for example, calculate the average number of loops per call. Another important piece of information is how often the two tests for the size of the result actually allow us to add a result to the remainder bound. Those branches are fairly expensive, so unless they actually reduce the number of elements to add up, we want to remove them.

Similar modifications were made to the multiplication code and other higher level operations where there is interesting information to collect. The counters are then output using a special command from COSY code, and the data can then be analyzed. With the information from this tool, it will be possible to decide wether a given feature in the code is actually increasing the performance, or if it slows down the algorithm.

4.2 Monitored Events

In Table 1 the events that are monitored if COSY was compiled with the PERFMON extension are listed. The number is the index into the `PERFMON` array used to store the statistics internally. This identification number is also printed when the data is output using the built in `PSTAT` procedure of COSY. If COSY was not compiled with PERFMON enabled, this command prints a warning message.

Number	Event	Function
1	Calls to function	ACCUMULATE
2	Outer loop iterations	ACCUMULATE
3	Inner loop iterations (i.e. number of Dekker additions)	ACCUMULATE
4	Results below cutoff in inner loop	ACCUMULATE
5	Results below cutoff in outer loop	ACCUMULATE
10	Calls to function	DEMDE
11	(reserved)	DEMDE
12	(reserved)	DEMDE
15	Calls to function	DESQR
16	(reserved)	DESQR
17	(reserved)	DESQR
20	Calls to function	DEMINV0
21	Total number of result limbs available	DEMINV0
22	Total number of result limbs used	DEMINV0
30	Calls to function	DESQRT0
31	Total number of Newton steps	DESQRT0
33	Number of argument reductions performed	DESQRT0
40	Calls to function	DEEXP0
41	Total number of orders evaluated	DEEXP0
50	Calls to function	DESINE0
51	Total number of orders evaluated	DESINE0
52	Total number of π argument reductions	DESINE
60	Calls to function	DELOG1
61	Total number of orders evaluated	DELOG1
62	Total number of argument reductions performed	DELOG0
70	Calls to function	DEATAN0
71	Total number of orders evaluated	DEATAN0
90	Calls to function	DESIGN
91	Number of reduction loops	DESIGN

Table 1: Monitored events and their respective id numbers.

5 Appendix

5.1 Compiler Options

For the Dekker operations, the exact order in which the floating point operations are executed, clearly is crucial. Great care has to be taken in choosing automatic compiler optimization options. Most compilers try to rearrange floating point operations, or use slightly less exact operations for operations like division or square roots, which are then not IEEE 754 compliant. All of these optimizations must to be avoided in our code, since our algorithms require full IEEE 754 compliance to be rigorous.

In the following sections we have listed some compiler options for compilers we use: Intel Fortran Compiler 9.2 (`ifort`) and GNU Fortran Compiler 4.2 (`gfortran`).

5.1.1 Intel Fortran Compiler 9.2

The Intel Compilers have very powerful optimization options, including automatic vectorization and parallelization of loops. They also have a tendency to reduce precision of floating point operations to increase speed.

The following compiler options control the optimization and code generation:

-O2 Turns on the most optimizations. This is the compiler default.

-ax* Selects the targeted CPU while still generating code that runs on all x86 CPUs. We use -axP to optimize for Intel Core Duo CPUs.

-ipo Does interprocedural optimizations on all source files at link time.

-fpconstant Interprets all floating point constants in the source code as double precision numbers. This switch is not necessary, since the COSY source code declares double precision constants explicitly.

-pad Allows the compiler to change the memory layout to make it fit to memory boundaries.

-nocheck Do not emit code for stack boundary checking at runtime.

-f77rtl Link with Intel's Fortran 77 compatible runtime libraries.

-fp-mode strict Tells the compiler to adhere strictly to the IEEE standard for floating point operations. According to the documentation, this guarantees correct rounding of all floating point operations in accordance with IEEE, as well as no reordering of floating point expressions.

-mp Deprecated switch with the same functionality as the -fp-model strict.

Our recommended command line options to compile COSY with this compiler are
`-O2 -axP -ipo -f77rtl -nocheck -fp-model strict -align`
which is also the command line used to compile the binary release versions of COSY Infinity[5].

[5] Except for minor adjustments in the target platform depending on the operating system.

5.1.2 GNU Fortran Compiler 4.3

We do not use this compiler frequently, therefor these suggestions should be checked before being used in production.

-O2 turns on most optimizations. According to the documentation, and our tests, no unsafe floating point optimizations are used at this setting.

-O3 Turn on full optimizations. However, at this level of optimization, we have had several problems. We do not recommend this setting.

Since we use Intel Fortran to compile COSY release versions, we do not have any recommended optimization parameters for the GNU Fortran Compiler.

References

[1] DoD supplement to ANSI X3.9-1978. Tech. rep., US Department of Defense, 1978.

[2] IEEE standard for binary floating-point arithmetic. Tech. Rep. IEEE Std 754-1985, The Institute of Electrical and Electronics Engineers, 1985.

[3] Braune, K., and Kramer, W. High-accuracy standard functions for intervals. In *Computer Systems: Performance and Simulation* (1986), M. Ruschitzka, Ed., vol. 2, Elsevier Science Publishers B.V., pp. 341–347.

[4] Bronstein, I. N., Semendjajew, K. A., Musiol, G., and Mühlig, H. *Taschenbuch der Mathematik*, 5th edition ed. Verlag Harri Deutsch, 2001.

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

[6] Goldberg, D. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv. 23*, 1 (1991), 5–48.

[7] Kahan, W. Pracniques: further remarks on reducing truncation errors. *Commun. ACM 8*, 1 (1965), 40.

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

[9] Makino, K., and Berz, M. COSY INFINITY version 9. *Nuclear Instruments and Methods 558* (2005), 346–350.

[10] Moore, R. E. Automatic local coordinate transformation to reduce the growth of error bounds in interval computation of solutions of ordinary differentical equations. In *Error in Digital Computation, Vol II* (1965), L. B. Rall, Ed.

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

[12] SHEWCHUK, J. R. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry 18* (1997), 305–363.