International Journal of Modern Physics A
Vol. 24, No. 5 (2009) 1019–1039

World Scientific
www.worldscientific.com

COMPUTATION OF HIGH-ORDER MAPS TO MULTIPLE MACHINE PRECISION

ALEXANDER WITTIG* and MARTIN BERZ†

Department of Physics and Astronomy, Michigan State University, Biomedical and Physical Sciences Bldg., East Lansing, MI 48824, USA

*wittigal@msu.edu

†berz@msu.edu

The Beam Dynamics simulation package in COSY INFINITY is built upon a differential algebra data type. With it, it is possible to compute transfer maps or arbitrary systems to arbitrary order. However, this data type is limited by the precision of the underlying floating point number model provided by the computer processor.

We will present a method to extend the effective precision of the calculations based purely on standard floating point operations. Those algorithms are then integrated into the differential algebra data type to efficiently extend the available precision, without unnecessarily affecting overall efficiency. To that effect, the precision of each coefficient is adjusted automatically during the calculation.

We will then proceed to show the effectiveness of our implementation by calculating high precision maps of combinations of homogeneous dipole segments, for which the exact results are known, and comparing the high precision coefficients with the results produced by the traditional COSY beam physics package.

Keywords: High precision; differential algebra; COSY INFINITY; high precision aberrations.

PACS numbers: 41.85.Ja, 41.85.Lc

1. Theory of High Precision Operations

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

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

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

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

Given any real number $r \in \mathbb{R}$ within the range of the floating point representation, we will denote by $\tilde{r} \in R$ the closest floating point number in the given system of floating point numbers. Then it follows readily from Definition 1.1 that

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

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

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

Definition 1.2. 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 the above definition, a bound for rounding errors and a useful condition for the mantissa of the result of a round-to-nearest operation $\odot$ easily follow. Let $z = m_z \cdot 2^{e_z} = a \odot b$. Then

$$|z - (a \bullet b)| < \epsilon_m \cdot z. \tag{1}$$

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

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

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

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

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,[2] 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 [2] 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 1.1. *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.*

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

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

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

To prove that $zz \in R_{53}$, we first note that we can write $w = i \cdot 2^{e_b}$ for some integer i since $e_a \geqslant e_b$. Secondly, we have that $|b - w| = |b - z + a| \leqslant |b|$ by optimality. To see this simply let $z = x$, and then apply Definition 1.2. We thus have

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

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

Note that by Definition 1.1 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 same two values without the condition that $a > b$, but requiring some additional floating point operations. The following algorithm is due to Knuth.[3] 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 1.2. *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, [3]. □

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 1.3. 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

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

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 $|[x] - x| \leqslant 1/2$, but depending on x it can have either positive or negative sign. So the missing bit is the sign bit of the tail of the splitting.

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

Theorem 1.3. *Let* $a \in R_{53}$ *be given and let* $p = x \otimes (2^{27} + 1)$. *Then the head of the splitting of* a *is given by* $a_h = p \oplus (x \ominus p)$.

Proof. Since the proof of this theorem is somewhat technical and does not contribute much to the understanding of these operations, we refer the reader to the papers of Dekker[2] or Shewchuk.[4] □

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 1.4. *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. □

1.3. *High precision numbers*

Based on the exact multiplication and addition, it is now possible to implement high precision numbers. Those numbers are stored as unevaluated sums of double precision floating point numbers. The value represented by that high precision number is given by the exact sum of all terms:

Definition 1.4. A high precision number a is given by a finite sequence of double precision floating point numbers a_i. We call each a_i a limb of the number. The value of a is given by

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

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

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

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

In order to make our high precision numbers rigorous, we add an error bound to the expansion, similarly to the remainder bound of Taylor Models.[5,6]

Definition 1.5. A high precision interval a is given by a high precision number consisting of n limbs a_i and a double precision error term a_{err}. The value of the

interval is then given by

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

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

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

1.3.1. *Accumulator*

The core operation used by our high precision numbers is the addition of a sequence of double precision numbers. The algorithm to perform that operation is called the accumulator. It takes a sequence of n double precision numbers a_i and returns another sequence of double precision numbers b_i of predefined maximum length. If there are roundoff errors or the result does not fit into the requested length, we optionally return an accumulated, outward rounded error term, which contains an upper bound on the error of the result. The input sequence is not required to be in any specific order, but to minimize the roundoff errors and to speed up execution time it is best if the input is sorted by decreasing order of magnitude. The output sequence is not guaranteed to be ordered, yet typically it will even be normalized, depending on the amount of cancellation happening during operations.

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

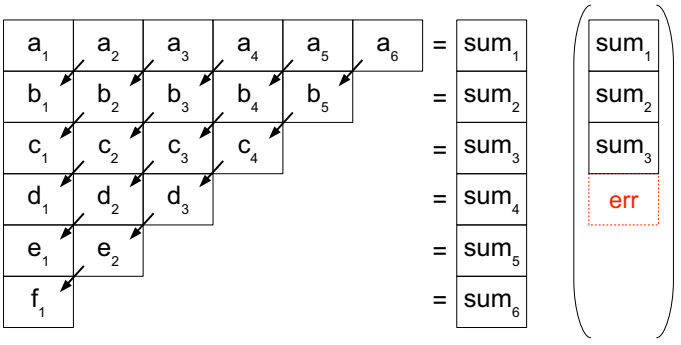

Fig. 1. The accumulator algorithm. In this example we want to add six double precision numbers $a_1 \ldots a_6$. The arrows indicate the error term of the exact addition of the two numbers above them. If only three limbs are desired for the result the summation terminates after three iterations and the left over terms are either discarded or accumulated into an outward rounded error term.

1.3.2. *High precision interval operations*

Utilizing the accumulator as outlined above, we have implemented a rigorous high precision interval arithmetic data type in COSY INFINITY.[7] As in any arithmetic, the two most important operations are addition and multiplication of two such intervals. Using the accumulator, addition is fairly straightforward. All we need to do is copy the limbs of each interval into an array and then send it through the accumulator. Since we get exact values for the leftover roundoff errors in this process, obtaining a remainder bound is trivial. For the multiplication we essentially use the fact that we can multiply each limb of the first argument with each limb of the second argument exactly using Dekker's algorithm. Then all the results of those multiplications are added up using the accumulator. Here, again, the accumulator gives us precise bounds for the left-over errors.

Division and intrinsic functions such as square roots, trigonometric functions, etc. are then implemented based on addition and multiplication. Using argument reduction based on mathematical identities, Taylor expansions, and Taylor remainder formulae for some intrinsics[8] or interval Newton iterations[9] for others, it is possible to calculate rigorous enclosures of the correct results. We will not go into more detail of this implementation, for a full discussion see [6].

At this point it is worth noting some differences between our implementation and other high precision implementations based on floating point expansions, such as the "double double" implementation in the original Dekker paper[2] and the arbitrary precision arithmetic by Shewchuk.[4] In those implementations the authors were very careful to develop algorithms that provide some guaranteed accuracy and derived error estimates for their operations. For floating point expansions, these estimates always lead to a requirement for normalization of the input. Normalization, however, is a rather complicated and computationally expensive operation. We, on the other hand, use an entirely different approach. Instead of making any analytic

estimates for roundoff errors, we have our numbers validate themselves by adding a rigorous remainder bound. The only claim we make is that the interval resulting from an operation rigorously encloses the mathematically correct result. If the input numbers are badly conditioned or if excessive cancellation occurs during the calculation, it is possible that our algorithm produces significant overestimation. The result, however, is rigorous nonetheless. Applying our high precision intervals to real world problems shows that those cases are very rare and that our intervals typically provide sharp enclosures of the correct results.[10] To the best of our knowledge, our implementation is the only rigorous interval implementation based on floating point expansions as of now.

2. Implementation of a High Precision Differential Algebra

Utilizing the same basic algorithms outlined above, we can begin to implement high precision DA vectors. Instead of starting from scratch, we will extend the already existing implementation of a DA vector in COSY. The final goal for this is to extend the existing DA vector type to high precision in such a way that it can act as a drop in replacement for the traditional double precision type. That is, old code should run without any major modification besides adding one command in the beginning of the code to select the desired precision. All subsequent operations should be transparently carried out without any change to the user interface. The full implementation of this is beyond the scope of this paper, but we will present the basic algorithms used.

As with the scalar data type, the two most important operations are addition and multiplication. Once these two have been properly implemented all other operations will follow naturally from those two. But before we can begin with the details of the implementation we shall introduce the way DA vectors are stored in our implementation.

2.1. *Storage*

Currently, DA vectors are stored as a list of double precision coefficients and an associated coding integer denoting the monomial for this coefficient. The method by which these coding integers are calculated is beyond the scope of this paper. For details, see [11]. Our way of representing high precision numbers blends in naturally with this storage format. Instead of only storing one double precision number per monomial, we will simply store several limbs continuously, all with the same coding integer. As with the high precision numbers, the value for each coefficient is the sum of all double precision numbers with the same coding integer.

By thus storing the limbs continuously in memory, we greatly simplify the addition and multiplication process as will be seen later. Also the typical output of such a DA vector is rather easy to interpret. As with the traditional DA vectors we simply print a list of all coefficients. Only this time there will not only be one

coefficient per monomial, but several. In the typical case the coefficients will decrease by orders of ϵ_m, so that the first coefficient printed should be roughly what the traditional DA vectors would produce, while all following coefficients will be corrections to that value. To make the output rigorous, we also print the exact binary representation of the floating point number as given in Definition 1.1.

2.2. *Addition*

The addition of two DA vectors is a fairly straightforward operation. It is a simple merging of the two vectors, where a monomial is copied into the output vector if it only appears in one of the vectors, or is added and then copied if a monomial appears in both vectors. In the high precision implementation we do exactly the same with the only difference that to add up two monomials appearing in both input vectors, we first copy the limbs into a temporary array, alternating between the two vectors. That way, as in the case of the high precision interval addition, we typically obtain an order of the array elements in roughly decreasing order of magnitude.

This array is then sent through the accumulator (see 1.3.1) that accumulates the numbers, starting with the smallest ones at the end of the temporary array, into the output vector. Once the maximum number of limbs has been reached, the absolute values of the left over terms are added up and kept in a separate tallying variable. This is in preparation for the implementation of Taylor Models, where all calculations are made rigorous by keeping an error bound.

This way of adding numbers is essentially the same we used for high precision scalars. Of course we thus get all the benefits that implementation had, including the automatic renormalization in case the input was denormalized and not too much cancellation occurs. Also the addition of a DA vector and a scalar (i.e. in COSY either a real or a high precision number) follows trivially from this. As always, once addition is implemented, by symmetry of the floating point numbers we also automatically have subtraction.

2.3. *Multiplication*

The more complex operation, and thus the core of our DA implementation, is the multiplication of two DA vectors. This is a very important operation that needs to be carried out quickly as it is used often throughout calculations. The performance of all other arithmetic operations depends crucially on this one operation. We therefore have taken great care to make this operation as fast as possible.

Because our way of storing high precision coefficients is quite natural in the current DA implementation, we can use most of the current implementation with only minor changes. A detailed description of the existing DA multiplication is beyond the scope of this paper, the reader is referred to [11] for full details. All we need to know here is that the algorithm walks through each coefficient in the first DA vector and multiplies it with every relevant coefficient in the second DA vector,

adding the result to a location in a temporary array which is determined by the coding integers of the first and the second coefficient. In order to determine which coefficients are "relevant", the algorithm first sorts the second DA vector by the monomial order of the coefficients. Note that this is not the same ordering induced by the coding integers.[11]

To amend this algorithm for efficient high precision operations, several things are necessary. First the temporary array which keeps all the new resulting coefficients needs to be enlarged, so it can keep the maximum number of limbs for each coefficient. Then we need to add the high precision multiplication. The naive way of achieving this would be to first extract all limbs for a given coefficient, and then to utilize the multiplication routine for high precision intervals. That, however, is very time consuming and also does not make use of some intrinsic properties of coefficients of DA vectors, which can be exploited to speed up the operation. So instead, we will rewrite the multiplication from scratch without relying on the high precision interval code here.

First, note that for the exact multiplication we need to split every coefficient into a head and a tail. This operation only has to be done once for each coefficient, and the most natural place to do the splitting is when the second DA vector is sorted by order. The first vector does not need to be pre-split since there every coefficient is only used once. Next, note that, instead of multiplying all limbs of the high precision coefficients at once, we can just multiply each limb separately and add the result to the appropriate resulting coefficient. There are two operations involved in this, a multiplication and then an addition.

One of the intrinsic properties of coefficients in DA vectors is that the magnitude of coefficients typically decrease exponentially with the order. We can use this fact to our advantage. In the COSY implementation of DA vectors, there is a cutoff value that specifies a lower bound on coefficients to be kept in the DA vector. Coefficients that are below that cutoff value are swept out of the calculation. We keep that concept of a cutoff value ϵ for our high precision implementation. Assuming that the high precision coefficients are normalized, we then can say that the last limb of any coefficient is of order ϵ/ϵ_m since all terms of order ϵ or lower are discarded. Now, when multiplying two coefficients, we do not immediately start with an expensive high precision multiplication. Instead, we first multiply the two coefficients in floating point arithmetic and then compare the order of the resulting number. If this product is of order ϵ/ϵ_m then the roundoff error due to floating point precision is of order $(\epsilon/\epsilon_m) \cdot \epsilon_m = \epsilon$ and thus is subject to the cutoff. In this case, there is no need to do any precise operations. We can proceed exactly as in the case of the traditional DA vectors, where the result is just added to the last limb of the correct monomial in the temporary array.

Only when this floating point multiplication yields a result that is of larger order than ϵ/ϵ_m do we perform a precise multiplication of the two coefficients. We still retain the approximate result of the floating point operation, though. This is because, in the next step, we need to add the two double precision numbers we

obtained from the exact multiplication to the resulting high precision coefficient. Naively, one could just copy the two numbers and all limbs in the result into an array and then send that through the accumulator. That, however, is not the most efficient way to perform the addition. Note that if we know the order of magnitude of the numbers we want to add, we can skip all limbs of higher order in the result since during the addition their value will not be affected. This is where the floating point product we calculated as a first step comes in. By estimating the order of magnitude of the approximate product, we can determine at which limb in the result we have to start to add the precise product.

The addition then also is carried out in an especially fast way. Assuming that the result is normalized, it is not necessary to add all numbers using the accumulator algorithm. Instead, we can do the following. Assume a_1 and a_2 are the two parts of the precise multiplication, and c_k is the limb of the result we determined to start our addition at. Then we perform an exact addition of a_1 and a_2 yielding $\tilde{a}$ and an error term a_{11}. Secondly, we exactly add $\tilde{a}$ and c_k to yield the new limb c_k and an error term a_{22}. Now we simply advance to the next limb of the result and repeat the same operations with a_{11}, a_{22} and c_{k+1}. We continue this process until we reach the maximum length of the result or until a_{11} and a_{22} are both zero.

A graphical representation of this algorithm is shown in Fig. 2. It is worth noting that at the end of this procedure, the two error terms left over are the exact roundoff error. Thus it is very easy to turn the DA vectors into rigorous Taylor Models later by simply adding the two left over terms to the correct tallying variable used in the rigorous Taylor Model operations.

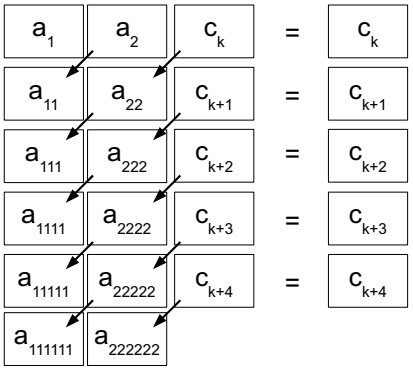

Fig. 2. Addition in the DA multiplication algorithm. a_1 and a_2 are the result of the exact addition, c_k is the limb of the result we start adding to. In this example there are four more limbs after that. The arrows indicate the error term of the exact addition of the two numbers above them.

Last we note that, since the limbs of a coefficient are stored linearly, we can keep the number of lookups in the coding integer tables constant. As in the traditional

DA vector case, only one lookup for each resulting monomial is required. To achieve that, we simply check if the coding integer for each coefficient is that same as that for the preceding one. If so, it is not necessary to perform another lookup since the last calculated address is still valid.

This completes the multiplication algorithm for high precision DA vectors.

To practically implement the algorithm outlined above, we have to have a way to estimate the order of magnitude of a floating point number. Fortunately, the representation of floating point numbers offers an easy way to do that. It is very fast to extract the exponent from a floating point number. But for the exponent e_a of a double precision floating point number a we have that

$$\ln_2(a) \leqslant e < \ln_2(a) + 1,$$

and thus it is a very good measure for the order of magnitude of a.

The efficiency of this implementation can be seen when comparing it to the traditional DA multiplication. Due to the floating point test we introduced, only operations which really need to be carried out in high precision are evaluated in that way. Practically this means that most higher order coefficients, which are typically very small, are treated exactly the same as before. Only the relatively few lower order coefficients are operated on in higher precision, but even here we try to keep the overhead small by only working on the relevant limbs. Thus the overhead is kept at a minimum.

2.4. *Other arithmetic operations*

The implementation of all other arithmetic operations from here on is straightforward. In fact, we can use the almost exact same code that already exists for the double precision DA vectors. The only somewhat non-trivial thing we need are the high precision intrinsic functions for high precision numbers. But as mentioned before, we already did implement a complete data type for high precision intervals based on the same representation used for the coefficients. So we can simply use those functions to calculate the needed high precision intrinsics.

The code for DA intrinsics essentially uses two methods: either a fixed point iteration or a simple Taylor expansion in the DA vector up to the necessary order to calculate the intrinsics. Without going into too much detail about the DA implementation, all intrinsics are reduced to one essential function. The intrinsics split off the constant part of the DA vector, evaluate the intrinsic on it using the corresponding high precision interval operation and then construct a one-dimensional polynomial of the non-constant remaining part of the DA argument. This is then evaluated efficiently by the procedure DAFUN. The original implementation of that function passes coefficients of the polynomial to evaluate as an array of double precision numbers. In the high precision implementation that clearly does not yield the desired result. The coefficients are instead passed as COSY variable numbers referencing high precision intervals. That is an easy and fast way to pass coefficients

to DAFUN while still allowing easy calculation of those coefficients in the calling intrinsic function. It also has the added benefit that repeating coefficients only require one COSY variable, as the same variable can be referenced several times as a coefficient. Those coefficients, of course, have to be calculated in high precision, otherwise the loss of precision there will lead to a loss of precision in the resulting DA vector and neutralize all benefits of high precision DA vectors.

Inside DAFUN we perform exactly the same operations as in the existing double precision implementation. Starting with the argument as the result we successively shift the highest remaining coefficient into the (empty) constant part of the result and multiply it by the argument using the high precision DA multiplication. We continue until all coefficients are used up. This is exactly the approach of the original implementation. The only difference is that we not only have to copy one but several double precision numbers into the result each time, as the coefficients themselves are high precision numbers consisting of several limbs. Since the DA vectors are non-verified, we can discard the high precision interval information and only use the interval's mid point as the coefficient.

With that change to DAFUN and minimal changes to all intrinsic functions to use high precision numbers for the coefficients, we can easily implement all remaining arithmetic operations. This for now completes the implementation of our basic high precision DA vectors. There are other functions in COSY which require further attention to make them work with the high precision DA vectors. Particularly the coefficient extraction operations to extract a certain coefficient from a DA vector need to be adjusted to extract not only one, but all limbs and to return a high precision number instead of just a double precision number.

2.5. *Output*

To output a high precision DA vector, we use essentially the same output format as for the traditional DA vectors. Non-zero coefficients of the expansion are printed, sorted by their order, each in one line. The first number in each line represents a running index of the coefficient, starting with 1. This is followed by a non-rigorous decimal fraction with 16 digits after the decimal point, representing an approximation to the coefficient's value. Note that, due to the nature of floating point numbers, not every floating point value can be represented as a decimal fraction of that length. The third field gives the order of that coefficient, followed by the exponents. Each integer in the exponent field represents a power of the corresponding independent variable.

Thus far, the output follows the traditional format. The last field in the line, however, is new. Here we print a rigorous representation of the floating point number. This is achieved by extracting the exact mantissa and exponent from the representation of the floating point number, as given by Definition 1.1. To assure the representation is compact, yet readable, we followed the representation of decimal fractions. Numbers are output as their mantissa value, the letter "b", followed by

the exponent. Similar to the letter "E" in the decimal fraction notation used in the first field, the "b" represents 2 raised to the power of the following integer.

Consider, for example, the following output line

```
6  -.1250000000000000       2   2 0                          -1b-3
```

This is the 6th line of the output, the decimal representation of the coefficient is given as -0.125. The coefficient has order two, and belongs to the $x^2y^0 = x^2$ term. The exact representation is given by $-1b-3 = -1 \cdot 2^{-3} = -\frac{1}{8}$. Although in this example the exact and decimal representation are the same, this is generally not true due to possible rounding in the conversion to a decimal fraction.

If the DA vector contains high precision coefficients, each limb of the coefficients is printed as a regular coefficient. Since the limbs are stored sequentially, they will also appear in sequence in the output. If the high precision coefficient is normalized, typically each limb will be about 10^{-15} times smaller than the previous limb. An example of this output style is given below.

```
1  0.8660254037844386       1   1 0        3900231685776981b-52
2  0.5017542110903451E-16   1   1 0        8141427543753187b-107
3  -.7479771237866948E-33   1   1 0       -8745358328802511b-163
4  0.2653073781648097E-49   1   1 0         87312899972227b-211
```

Note how each line has the same exponents, $x^1y^0 = x$, thus signaling that these coefficients represent limbs of the same high precision coefficient. The correct value for this coefficient is the exact sum of all limbs. To calculate a decimal representation of the coefficient, one can simply add up the exact representation of each limb using a third party high precision package.

3. Illustrating Example

The DA vectors in COSY INFINITY were initially implemented to derive arbitrary order transfer maps for beam physics applications.[12] This is particularly useful in connection with assessment of long-term stability through symplectic tracking.[13,14] To show how our high precision DA vectors can help to get even better results, we will present transfer maps for several homogeneous dipole magnet segments.

Currently we cannot yet just run the existing COSY INFINITY code with high precision DA vectors due to the missing implementation of some essential functions used in the COSY beam physics package. Instead, we have manually derived an algebraic equation to calculate the final coordinates x_f and a_f of a beam depending on the initial position x_i and slope a_i in a homogeneous dipole magnet of reference radius r_0 and angle θ based on the geometry of the problem.

Let

$$c_x = r_0 \sin(a_i),$$

$$c_y = x_i + r_0 \left(1 - \cos(a_i)\right),$$

$$A = \tan(\theta)^2 + 1,$$

$$B = 2\left(\tan(\theta) c_x + c_y\right),$$

$$C = c_x^2 + c_y^2 - r_0^2.$$

Then

$$x_f = \frac{-B + \sqrt{B^2 - 4 \cdot A \cdot C}}{2\sqrt{A}} - r_0, \tag{3}$$

$$a_f = \theta - \arctan\left(\frac{r_0 \sin(a_i) - (x_f + r_0)\sin(\theta)}{x_i + r_0(1 - \cos(a_i)) - (x_f + r_0)\cos(\theta)}\right). \tag{4}$$

The corresponding COSY INFINITY function that represents the same beam line element is `MS` r_0 θ `1 0 0 0 0 0;`.

With our high precision DA vector implementation we should, of course, be able to reproduce essentially the same map as the COSY beam code, except for small roundoff errors. Our test case will be a homogeneous dipole segment of angle $\theta = \frac{\pi}{6}$ and reference radius $r_0 = 1$. The expansion for the final position and slope as calculated by the current COSY INFINITY double precision code using the command

```
MS 1 30 1 0 0 0 0 0;
```

is given in Fig. 3. Comparing that to the results of the evaluation of Eq. (3) in high precision DA arithmetic with about 60 digits, given in Fig. 4, shows that, indeed, the leading limb of the result is the same as in the COSY beam physics expansion. All the small roundoff errors of magnitude 10^{-15} encountered in the double precision expansion, however, are not seen in the high precision implementation. We have set the cutoff value, below which coefficients are discarded, to 10^{-60} in the high precision operation. That is why there seem to be no roundoff errors in the high precision map. They were all below the cutoff value.

Note how some of the coefficients actually do not require high precision. At first glance it looks like there is no benefit to high precision calculations. All there is, is the same number as in the double precision case. But actually this result shows how well our implementation works. Those coefficients can be accurately represented by only one double precision number. All digits up to 10^{-60} are actually 0.

To see the full power of the high precision DA vectors, we now combine 12 of those 30 degree segments together to form a full 360 degree bend. Clearly the resulting transfer map has to be the identity. Simply repeating the COSY `MS` command 12 times yields the expected transfer map shown in Fig. 5. The linear coefficients are almost 1 whereas the other coefficients are all of order 10^{-15} or less. On the other hand, iterating Eq. (3) 12 times with high precision DA vectors results in the map shown in Fig. 6. In these calculations, we set the cutoff value to 10^{-70}. That way, the small roundoff error terms become visible in the result. As before the result is the identity map, up to roundoff noise introduced during the calculation. The notable difference is that, since we work with high precision numbers of length 60, the noise is of the order of 10^{-63}.

x_f:

```
 I  COEFFICIENT             ORDER EXPONENTS
 1  0.8660254037844386        1   1 0  0
 2  0.4999999999999999        1   0 1  0
 3  -.1249999999999999        2   2 0  0
 4  0.4330127018922192        2   1 1  0
 5  0.5801270189221930E-01    2   0 2  0
 6  -.1250000000000000        3   1 2  0
 7  0.2165063509461096        3   0 3  0
 8  -.7812499999999993E-02    4   4 0  0
 9  0.5412658773652738E-01    4   3 1  0
10  -.1406249999999999        4   2 2  0
11  0.1623797632095823        4   1 3  0
12  0.6690675473054825E-02    4   0 4  0
13  -.1562499999999999E-01    5   3 2  0
14  0.8118988160479110E-01    5   2 3  0
15  -.1718749999999999        5   1 4  0
16  0.1353164693413186        5   0 5  0
-------------------------------------
```

a_f:

```
 I  COEFFICIENT             ORDER EXPONENTS
 1  -.4999999999999999        1   1 0  0
 2  0.8660254037844386        1   0 1  0
 3  -.2500000000000000        2   0 2  0
 4  -.6250000000000000E-01    4   0 4  0
 5  0.2220446049250313E-15    5   2 3  0
 -------------------------------------
```

Fig. 3. Transfer map for a 30 degree homogeneous dipole with COSY beam package.

x_f:

I	COEFFICIENT	ORDER	EXPONENTS	EXACT REPRESENTATION
1	0.8660254037844386	1	1 0	3900231685776981b-52
2	0.5017542110903451E-16	1	1 0	8141427543753187b-107
3	-.7479771237866948E-33	1	1 0	-8745358328802511b-163
4	0.2653073781648097E-49	1	1 0	87312899972227b-211
5	0.5000000000000000	1	0 1	1b-1
6	-.1250000000000000	2	2 0	-1b-3
7	0.4330127018922193	2	1 1	3900231685776981b-53
8	0.2508771055451726E-16	2	1 1	8141427543753187b-108
9	-.3739885618933474E-33	2	1 1	-8745358328802511b-164
10	0.1326536890824049E-49	2	1 1	87312899972227b-212
11	0.5801270189221933E-01	2	0 2	2090127860996437b-55
12	-.2667865061111657E-17	2	0 2	-6926173687902441b-111
13	0.1119742698409979E-34	2	0 2	4189454815015701b-168
14	-.9845464222053095E-52	2	0 2	-2592120999413b-214
15	-.1250000000000000	3	1 2	-1b-3
16	0.2165063509461096	3	0 3	3900231685776981b-54
17	0.1254385527725863E-16	3	0 3	8141427543753187b-109
18	-.1869942809466737E-33	3	0 3	-8745358328802511b-165
19	0.6632684454120243E-50	3	0 3	87312899972227b-213
20	-.7812500000000000E-02	4	4 0	-1b-7
21	0.5412658773652741E-01	4	3 1	3900231685776981b-56
22	0.3135963819314657E-17	4	3 1	8141427543753187b-111
23	-.4674857023666842E-34	4	3 1	-8745358328802511b-167
24	0.1658171113530080E-50	4	3 1	21828224993057b-213
25	-.1406250000000000	4	2 2	-9b-6
26	0.1623797632095823	4	1 3	45705840067699b-48
27	-.4469896349870486E-17	4	1 3	-1450564298463051b-108
28	0.2449402781674419E-33	4	1 3	2863844940720025b-163
29	0.2101110160114334E-49	4	1 3	138295453756081b-212
30	0.6690675473054831E-02	4	0 4	7713823633230503b-60
31	0.2003954727104893E-18	4	0 4	2081025566838551b-113
32	0.2799356746024948E-35	4	0 4	4189454815015701b-170
33	-.2461366055511375E-52	4	0 4	-2592120999411b-216
34	-.1562500000000000E-01	5	3 2	-1b-6
35	0.8118988160479113E-01	5	2 3	45705840067699b-49
36	-.2234948174935243E-17	5	2 3	-1450564298463051b-109
37	0.1224701390837209E-33	5	2 3	2863844940720025b-164
38	0.1050555080057157E-49	5	2 3	553181815024319b-215
39	-.1718750000000000	5	1 4	-11b-6
40	0.1353164693413186	5	0 5	4875289607221227b-55
41	-.1297677216343504E-16	5	0 5	-4211203333632873b-108
42	-.3094644200303947E-33	5	0 5	-7236524105121281b-164
43	0.1750925133428610E-49	5	0 5	3687878766828823b-217

Fig. 4. Transfer map for a 30 degree homogeneous dipole with high precision DA vectors (normalized).

a_f:

```
 I  COEFFICIENT            ORDER EXPONENTS        EXACT REPRESENTATION
 1  -.5000000000000000       1   1 0                          -1b-1
 2  0.8660254037844386       1   0 1           3900231685776981b-52
 3  0.5017542110903451E-16   1   0 1           8141427543753187b-107
 4  -.7479771237866948E-33   1   0 1          -8745358328802511b-163
 5  0.2653073781648059E-49   1   0 1            349251599888903b-213
 6  -.2500000000000000       2   0 2                          -1b-2
 7  -.6250000000000000E-01   4   0 4                          -1b-4
 ----------------------------------
```

Fig. 4. (*Continued*)

x_f:

```
 I  COEFFICIENT            ORDER EXPONENTS
 1  0.9999999999999993       1   1 0  0
 2  -.1665334536937735E-15   1   0 1  0
 3  -.1665334536937734E-15   2   1 1  0
 4  0.3330669073875470E-15   2   0 2  0
 5  0.5499073418846478E-15   4   2 2  0
 6  0.3348016308635238E-15   4   1 3  0
 7  0.1942890293094024E-15   4   0 4  0
 8  -.1387778780781446E-15   5   4 1  0
 9  -.1110223024625157E-15   5   3 2  0
10  0.1110223024625157E-15   5   2 3  0
11  0.1082467449009528E-14   5   1 4  0
 ---------------------------------------
```

a_f:

```
 I  COEFFICIENT            ORDER EXPONENTS
 1  0.1665334536937735E-15   1   1 0  0
 2  0.9999999999999993       1   0 1  0
 3  -.2220446049250313E-15   4   1 3  0
 4  -.7359851755371739E-15   5   4 1  0
 5  0.9601694439531627E-15   5   3 2  0
 6  0.1214306433183765E-14   5   2 3  0
 7  0.5273559366969494E-15   5   1 4  0
 ---------------------------------------
```

Fig. 5. Transfer map for 12 30 degree homogeneous dipoles with COSY beam package.

x_f:

I	COEFFICIENT	ORDER	EXPONENTS	EXACT REPRESENTATION
1	1.000000000000000	1	1 0	1b0
2	0.1001967514747971E-62	1	0 1	3712637180178849b-261
3	-.8517491040037048E-63	2	2 0	-3156025864275235b-261
4	0.8502528978208232E-63	2	1 1	6300963802800349b-262
5	-.8058934833360440E-63	2	0 2	-5972229768846003b-262
6	-.1421155290465671E-62	3	3 0	-164558540213685b-256
7	-.1914009193660878E-62	3	2 1	-7092067947309569b-261
8	-.3175427147967325E-62	3	1 2	-367689343684029b-256
9	0.1793778628155982E-62	3	0 3	3323285999734849b-260
10	0.3216870906455098E-62	4	4 0	372488203065175b-256
11	-.7424413638917452E-63	4	3 1	-5502005546318573b-262
12	-.2598190020076057E-62	4	2 2	-4813597610562419b-260
13	0.2515338308647966E-62	4	1 3	4660100446352085b-260
14	0.1805036934614915E-62	4	0 4	6688287929906621b-261
15	0.8011932350143374E-63	5	5 0	5937397668167309b-262
16	-.7490211353162293E-62	5	4 1	-6938457771293817b-259
17	-.7799594535945731E-62	5	3 2	-7225050772168895b-259
18	0.4235531564115624E-62	5	2 3	7847056781432733b-260
19	-.1500571371277421E-62	5	1 4	-5560137412157345b-261
20	0.1584098366644266E-63	5	0 5	4695707121437689b-264

a_f:

I	COEFFICIENT	ORDER	EXPONENTS	EXACT REPRESENTATION
1	0.3560837904659925E-63	0	0 0	75b-217
2	-.2278936258982352E-63	1	1 0	-3b-213
3	1.000000000000000	1	0 1	1b0
4	0.1367361755389411E-62	1	0 1	9b-212
5	0.7976276906438231E-63	2	2 0	21b-214
6	0.1583069429958924E-63	2	1 1	2346328533794189b-263
7	-.6077163357286271E-63	2	0 2	-1b-210
8	-.1367361755389411E-62	3	3 0	-9b-212
9	0.1411139791891778E-62	3	2 1	2614381195344969b-260
10	-.2765903754149412E-62	3	1 2	-5124316389156745b-260
11	-.2051042633084117E-62	3	0 3	-27b-213
12	-.1139468129491176E-63	4	4 0	-3b-214
13	0.2033073242981725E-62	4	3 1	941655193509359b-258
14	0.2449856478406028E-62	4	2 2	129b-215
15	0.9779722569634157E-63	4	1 3	7247452854395351b-262
16	0.1185759022251755E-62	4	0 4	8787296929185793b-262
17	-.4329978892066468E-62	5	5 0	-57b-213
18	0.7659812972263426E-62	5	4 1	7095565977803831b-259
19	0.5849269731388036E-62	5	3 2	77b-213
20	0.1000543052127116E-61	5	2 3	4634198815107191b-258
21	-.5147305191874726E-62	5	1 4	-2384068888437037b-258
22	0.2468847613897548E-62	5	0 5	65b-214

Fig. 6. Transfer map for 12 30 degree homogeneous dipoles with high precision DA vectors (normalized).

References

1. IEEE standard for binary floating-point arithmetic. Technical Report IEEE Std 754-1985, The Institute of Electrical and Electronics Engineers, 1985.
2. T.J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 1971/72.
3. D. E. Knuth. *The Art of Computer Programming*, volume I-III. Addison Wesley, Reading, MA, 1973.
4. J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18:305–363, 1997.
5. K. Makino. *Rigorous Analysis of Nonlinear Motion in Particle Accelerators*. PhD thesis, Michigan State University, East Lansing, Michigan, USA, 1998. Also MSUCL-1093.
6. A. Wittig and M. Berz. Design and implementation of a high precision arithmetic with rigorous error bounds. Technical Report MSUHEP 081126, Michigan State University, 2008.
7. K. Makino and M. Berz. COSY INFINITY version 9. *Nuclear Instruments and Methods*, 558:346–350, 2005.
8. K. Braune and W. Kramer. High-accuracy standard functions for intervals. In Manfred Ruschitzka, editor, *Computer Systems: Performance and Simulation*, volume 2, pages 341–347. Elsevier Science Publishers B.V., 1986.
9. R. E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, NJ, 1966.
10. A. Wittig, M. Berz, and S. Newhouse. Computer assisted proof of the existence of high order fixed points. In *Proceedings, Fifth Int. Workshop on Taylor Model Methods, Toronto, Canada*, 2008.
11. M. Berz. Forward algorithms for high orders and many variables. *Automatic Differentiation of Algorithms: Theory, Implementation and Application*, SIAM, 1991.
12. M. Berz. *Modern Map Methods in Particle Beam Physics*. Academic Press, San Diego, 1999. Also available at http://bt.pa.msu.edu/pub.
13. B. Erdlyi and M. Berz. Optimal symplectic approximation of Hamiltonian flows. *Physical Review Letters*, 87,11:114302, 2001.
14. B. Erdlyi and M. Berz. Local theory and applications of extended generating functions. *International Journal of Pure and Applied Mathematics*, 11,3:241–282, 2004.