The Design and Implementation of a Rigorous High Precision Floating Point Arithmetic for Taylor Models

Alexander Wittig

Department of Physics, Michigan State University
East Lansing, MI, 48824

4th International Workshop on Taylor Methods
Boca Raton, 2006

Outline

Why do we need high precision?

- Standard floating point numbers only have a limited precision. In the case of double length FPs, for example, this means that the ratio $\frac{\delta r}{r} \approx 10^{-16}$.
- This is not enough for certain applications. Computer aided mathematical proofs like Johannes Grote is doing them sometimes need to get closer to the correct result.
- Also COSY GO Kyoko Makino presented yesterday needs higher precision to find the minima of a function even more precisely.

Floating Point Representation (IEEE 754)

- The representation of floating point numbers used in most modern computer hardware is defined in the standard IEEE 754.
- They are basically stored in a representation like $S \cdot M \cdot 2^E$ where S is the sign (either $+$ or $-$), M is the mantissa and E is the exponent. The precision of such a number is simply determined by the number of bits in the mantissa.
- IEEE 754 defines doubles as effectively having a 53 bit mantissa and an 11 bit exponent.

Design Goals

We have three main design goals for this arithmetic:

Rigorous	Since we want to use these algorithms in our verified Taylor models we have to make sure to provide an error bound that encloses all floating point roundoff errors that are made during the calculation process.
Fast	If this code is to be integrated into COSY we need it to be fast enough to meet the high COSY standards.
Adaptive	In the end we would like to be able to choose the precision adaptively so that we can represent the higher order terms in the Taylor model by lower precision, since they are small anyway.

Rigorous Arithmetic

To use these algorithms for Taylor models we need to be able to produce a verified enclosure of the correct result. This has several consequences for our algorithm:

- Whenever we do calculations we need to make sure that we take all possible round off errors into account and add them to an error bound.
- On the other hand we do not have to be absolutely precise in our calculations as long as we can give an accurate remainder bound for the result.

High Precision using Floating Point Arithmetic

- To get a higher precision we want to store numbers as *unevaluated sums* of doubles.
- The number of terms in this unevaluated sum determines the achievable precision and is called the *length* of the number
- Thus a high precision number A is represented as

$$A = \sum_i a_i$$

where $\{a_1, a_2, \cdots, a_i\}$ are the components that are stored as a list of doubles in memory.

High Precision using Floating Point Arithmetic

We call a number *normalized*, if

$$|a_{i+1}| < \epsilon \cdot |a_i| \; \forall a_i$$

where ϵ is the machine precision.
By requiring our numbers to be normalized we can simplify the algorithms we use later quite a bit.

Exact Addition and Dekker's Algorithms

To actually carry out calculations with these numbers we use some basic algorithms. The exact addition has been published by D. Knuth, while the multiplication was published by T. J. Dekker.

- They allow us to add or multiply two floating point numbers in such a way that we get two floating point numbers that again form an unevaluated sum of the correct result.
- They only need floating point operations that are IEEE 754 compliant and do not rely on any special features of some platform.

Exact Addition

The basic steps in an exact addition of two doubles a and b into r_1 and r_2. Here $\boxplus$ and $\boxminus$ are the floating point operations:

1. Set $r_1 = a \boxplus b$. This already is the first part of the result.
2. Set $b_{virt} = x \boxminus a$
3. Set $a_{virt} = x \boxminus b_{virt}$
4. Set $r_2 = (b \boxminus b_{virt}) \boxplus (a \boxminus a_{virt})$

Dekker Multiplication

The Dekker Multiplication of a and b is a bit more involved. Here is an outline of how it works:

1. Split up a and b into sums of two “half length” numbers, called *head* and *tail* each of which has only half of its mantissa bits set.
2. Multiply both heads, and each head with each tail.
3. Add the two cross terms. Then add the head product to this using exact addition. The first term of the addition result is also the first term of the multiplication result.
4. Add the tail product to the second term of the addition. This is the second term of the result.

Quadruple Length Precision

First we want to implement “Quadruple Precision” addition and multiplication. That means for now each high precision number consists of two double precision numbers.
I will show two versions of the algorithms:

- A non-rigorous one, which will demonstrate the basic principle
- And a rigorous version, which will actually take all the errors we make into account and sum them up in an error interval.

Since COSY is written in FORTRAN those algorithms will also be written in FORTRAN as well, so they can be easily integrated into our current code base.

Quadruple Length Precision

In the following slides I will use this notation:

- The boxed operators $\boxplus, \boxminus, \boxtimes$ refer to floating point operations
- $\oplus$ and $\otimes$ represent the exact operations.
- The regular operators $+, -, \cdot$ are the normal mathematically exact operations.
- There are three quadruple length numbers: $A = (a_1, a_2, a_{err})$, $B = (b_1, b_2, b_{err})$ and $C = (c_1, c_2, c_{err})$. The error parts are only used in the rigorous part.
- We will assume that A and B are normalized

Quadruple Length Addition

To calculate $C = A + B$ we only have to do the following:

QuadrupleAdd(A, B)

1. $(c_1, c_2) = a_1 \oplus b_1$
2. $c_2 = c_2 \boxplus a_2 \boxplus b_2$

Quadruple Length Addition

To calculate $C = A + B$ we only have to do the following:

```
QuadrupleAdd( A, B )
1  $(c_1, c_2) = a_1 \oplus b_1$
2  $c_2 = c_2 \boxplus a_2 \boxplus b_2$
```

1. Exact addition of the biggest two components.

Quadruple Length Addition

To calculate $C = A + B$ we only have to do the following:

```
QuadrupleAdd( A, B )
1  $(c_1, c_2) = a_1 \oplus b_1$
2  $c_2 = c_2 \boxplus a_2 \boxplus b_2$
```

1. Exact addition of the biggest two components.
2. Use regular floating point operations to add up the rest. Here it does not make sense to use expensive exact addition because we would throw away the smaller part anyway!

Quadruple Length Multiplication

To calculate $C = A \times B$ we only have to do the following:

QuadrupleMult(A, B)

1. $(c_1, c_2) = a_1 \otimes a_2$
2. $c_2 = c_2 \boxplus (a_1 \boxtimes b_2) \boxplus (a_2 \boxtimes b_1)$

Quadruple Length Multiplication

To calculate $C = A \times B$ we only have to do the following:

QuadrupleMult(A, B)

```
1  $(c_1, c_2) = a_1 \otimes a_2$
2  $c_2 = c_2 \boxplus (a_1 \boxtimes b_2) \boxplus (a_2 \boxtimes b_1)$
```

1. Multiply the two biggest terms exactly.

Quadruple Length Multiplication

To calculate $C = A \times B$ we only have to do the following:

QuadrupleMult(A, B)

1. $(c_1, c_2) = a_1 \otimes a_2$
2. $c_2 = c_2 \boxplus (a_1 \boxtimes b_2) \boxplus (a_2 \boxtimes b_1)$

1. Multiply the two biggest terms exactly.
2. Add the cross terms to the correction term obtained above. Again we do not need to do this using exact multiplication because the error term would be thrown away anyway!

Quadruple Length Multiplication

To calculate $C = A \times B$ we only have to do the following:

```
QuadrupleMult( A, B )
1  $(c_1, c_2) = a_1 \otimes a_2$
2  $c_2 = c_2 \boxplus (a_1 \boxtimes b_2) \boxplus (a_2 \boxtimes b_1)$
```

1. Multiply the two biggest terms exactly.
2. Add the cross terms to the correction term obtained above. Again we do not need to do this using exact multiplication because the error term would be thrown away anyway!

Note that we do not even calculate $a_2 \boxtimes b_2$ because with normalized input this will be below our precision.

Rigorous Quadruple Length Addition

Now let's look at the rigorous error handling. To calculate $C = A + B$ rigorously, we have to do some modifications to the code:

QuadrupleAddR(A, B)

```
1  $(c_1, c_2) = a_1 \oplus b_1$
2  $c_2 = c_2 \boxplus a_2$
3  $c_{err} = (\epsilon \boxtimes c_2)$
4  $c_2 = c_2 \boxplus b_2$
5  $c_{err} = (c_{err} \boxplus (\epsilon \boxtimes c_2) \boxplus a_{err} \boxplus b_{err}) \boxtimes 2$
```

Rigorous Quadruple Length Multiplication

QuadrupleMultR(A, B)

```
1. $(c_1, c_2) = a_1 \otimes b_1$
2. $temp = a_2 \boxtimes b_1$
3. $c_{err} = (\epsilon \boxtimes temp)$
4. $c_2 = c_2 \boxplus temp$
5. $c_{err} = c_{err} \boxplus (\epsilon \boxtimes c_2)$
6. $temp = a_1 \boxtimes b_2$
7. $c_{err} = c_{err} \boxplus (\epsilon \boxtimes temp)$
8. $c_2 = c_2 \boxplus temp$
9. $c_{err} = (c_{err} \boxplus (\epsilon \boxtimes c_2) \boxplus (a_{err} \boxtimes |b_1|) \boxplus (a_{err} \boxtimes |b_2|) \boxplus (b_{err} \boxtimes |a_1|) \boxplus (b_{err} \boxtimes |a_2|) \boxplus (a_{err} \boxtimes b_{err})) \boxtimes 2$
```

Further steps

Further areas that we will have to look into are:

1. Make output of our functions normalized again.
2. Compilers and platform dependence.
3. Extension to higher precision than just double double.
4. Integration into the current COSY Taylor model code.
5. Implementation of input/output from/to ASCII representation.

Compilers and platforms

As we have heard we need several preconditions for our algorithms to work:

1. IEEE 754 compliance:
   Here we right now rely on our Intel Fortran compiler flags to force floating point operations to be standard compliant (`-mp`). At some later point we will probably set this using functions from the F90 standard at the start of the COSY program.
2. Round to nearest:
   Round to nearest is the rounding mode we need for our multiplication. Right now, again, we set this at compile time using a compiler flag and rely on the compiler to do it right. Later we will again do that at the start up of COSY using F90 functions.

Extension to higher precision

Our needs for high speed have to be combined with existing high precision libraries.

1. For "low" high precision, such as double-double or possibly triple-double we need to implement operations directly in COSY since we can't use the C libraries directly due to the overhead of function calls.
2. For higher precisions we can use existing libraries since here the cost of additional computation will outweigh the cost of the function call.

Integration into COSY: Precision

- Taylor models have only few large coefficients in the low order terms, but many small ones in the higher order. Therefore we will not store all coefficients with the same precision, but only those that really need to be high precision.
- By fixing a targeted remainder bound size for the whole Taylor model, COSY should automatically adapt the precision of the coefficients based on that cutoff. Thus we can probably reduce the performance impact significantly.

Integration into COSY: Storage

- In COSY Taylor models are stored as a sequence of doubles and their “address”, i.e. their order.
- It is very easy to just store several doubles for one address and just interpret them as the unevaluated sums needed for our high precision approach.
- This way most of the COSY Taylor model implementation as it exists now will remain unchanged.

Rigorous Input/Output

- At some point we surely want to output the result in human readable form. For this we need rigorous output and a way to specify the number of digits to be printed.
- Since in COSY code you still want to use decimal constants as you do now, we also need a converter to parse strings into high precision numbers.
- The problem with both is the rigorous conversion from the binary storage format to the decimal output format and vice versa. We have to make sure, that if the output is somehow inexact, we add the uncertainty to the remainder bound.
- Machine readable exact output could be done as hex digits encoding the bit pattern of the doubles. So you can store a full Taylor model in a file and read it again without any loss of precision due to rounding during I/O.

Thank you.

Thank you for your attention.