High precision evaluation of nonlinear functions

Siegfried M. Rump

Inst. f. Computer Science III
Hamburg University of Technology
Schwarzenbergstrasse 95
21071 Hamburg, Germany
Email: rump@tu-harburg.de

Abstract—A key to the accurate solution of an ill-conditioned system of nonlinear equations is the accurate evaluation of the functions is use. For this purpose we present a fast method to emulate quadruple precision for the basic operations as well as for the elementary standard functions. Our methods are based on a recent paper on accurate evaluation of sums and dot products, which in turn uses so-called error-free transformations. We use only double precision floating point operations, and the code contains no branches or access to mantissa or exponent. Hence the algorithms are fast in terms of *measured* computing time.

1. Introduction

Let floating point numbers and binary arithmetic according to IEEE 754 arithemtic [2] be given. Denote the set of double precision floating point numbers by $\mathbb{F}$, and let $\mathrm{fl}(\cdot)$ denote the result of a floating point computation, where all operations inside the parentheses are executed in double precision. It is well known that the correction of floating point addition and multiplication is *exact*. That is

$$a, b \in \mathbb{F},\ x = \mathrm{fl}(a + b) \ \Rightarrow\ y = a + b - x \in \mathbb{F}. \qquad (1)$$

Already in 1969, Knuth [3] presented a simple algorithm to compute the correction y:

function $[x, y] =$ `TwoSum`(a, b)
$x = \mathrm{fl}(a + b)$
$z = \mathrm{fl}(x - a)$
$y = \mathrm{fl}((a - (x - z)) + (b - z))$

The algorithm requires 6 flops *without* branch. A similar algorithm for multiplication has been given by G.W. Veltkamp (see [1]):

function $[x, y] =$ `TwoProduct`(a, b)
$x = \mathrm{fl}(a \cdot b)$
$[a_1, a_2] =$ `Split`(a)
$[b_1, b_2] =$ `Split`(b)
$y = \mathrm{fl}(a_2 \cdot b_2 - (((x - a_1 \cdot b_1) - a_2 \cdot b_1) - a_1 \cdot b_2))$

This algorithm uses the ingenious method by Dekker [1] to split a 53-bit double precision number into two 26-bit parts:

function $[x, y] =$ `Split`(a)
$c = \mathrm{fl}(\texttt{factor} \cdot a)$ % `factor` $= 2^{27} + 1$
$x = \mathrm{fl}(c - (c - a))$
$y = \mathrm{fl}(a - x)$

The results of this algorithm satisfy $x + y = a$. The trick is that the sign is used as the additional bit. Note that the multiplication of 26-bit floating point numbers is exact in double precision.

In [7] we used these error-free transformations to develop algorithms for summation and dot product producing a result *as if* computed in quadruple precision but using only double precision operations. Similar algorithms with result of k-fold precision are presented as well. All algorithms are without branch. The quadruple dot product algorithm is 40 % faster than the corresponding "state-of-the-art" XBLAS algorithm [4].

Here we develop quadruple precision algorithms for the four basic arithmetic operations and for elementary standard functions. Again no special operations such as access to mantissa or exponent are necessary, hence our algorithms are fast in terms of measured computing time. A Matlab toolbox [5] defining a `quad` data type has been written to allow easy access to high precision operations.

Based on [7] our algorithms are easily extended to producing results of k-fold precision.

2. High precision operations

We represent quadruple precision floating point numbers by a pair $[x, y]$ of double precision numbers $x, y \in \mathbb{F}$, and we denote the set of `quad` or extended precision numbers by $\mathbb{E}$.

In [7] we developed the following algorithm to calculate the sum of a vector of double precision floating point numbers. The result satisfies an accuracy estimation *as if calculated* in quadruple precision and then *rounded to double precision*. The following version is adapted to omit the rounding to double but rather to give back a result of

In Proceedings of 2005 International Symposium on Nonlinear Theory and its Applications, Bruge, Belgium, October 18–21, pages 733–736, 2005

quadruple precision.

$$\begin{array}{l}\text{function } [x,y] = \texttt{Sum2}(p)\\ \quad \text{for } i = 2:n\\ \quad\quad [p_i, p_{i-1}] = \texttt{TwoSum}(p_i, p_{i-1})\\ \quad x = p_n\\ \quad y = \mathrm{fl}\left(\sum_{i=1}^{n-1} p_i\right)\end{array}$$

Note that we use the Matlab convention that input and output of a routine are strictly separated. In [7, Proposition 4.5 and (4.11)] we proved the following result. Due to the fact that TwoSum is an error-free transformation, the sum $\sum p_i$ of the original vector p and the transformed vector are identical.

Theorem 1 *Suppose Algorithm* `Sum2` *is applied to floating point numbers $p_i \in \mathbb{F}$, $1 \le i \le n$, set $s := \sum p_i \in \mathbb{R}$ and $S := \sum |p_i|$, and suppose* $n\mathrm{eps} < 1$. *Then, also in the presence of underflow,*

$$|x + y - s| \le \gamma_{n-2}\gamma_{n-1} S, \tag{2}$$

where $\mathrm{eps} = 2^{-53}$ *denotes the relative rounding error unit and* $\gamma_k := k\mathrm{eps}/(1 - k\mathrm{eps})$.

For given two quad numbers $A = [A_{hi}, A_{lo}]$ and $B = [B_{hi}, B_{lo}]$ this already solves the problem of addition and subtraction by summing the vectors $[A_{hi}, A_{lo}, B_{hi}, B_{lo}] \in \mathbb{F}^4$ and $[A_{hi}, A_{lo}, -B_{hi}, -B_{lo}] \in \mathbb{F}^4$ using `Sum2`. The result $[x, y]$ satisfies

$$|A + B - (x + y)| \le 5\mathrm{eps}^2/(1 - 5\mathrm{eps}) ,$$

which basically means quadruple precision.

Multiplication in quad precision could be solved using `Split` and `Sum2`. This is roughly the way XBLAS proceeds for dot products. However, in [7, Algorithm 5.3 (`Dot2`)] we proposed a faster way by treating the lower order terms in a better way. The adapted algorithm for the dot product is as follows.

$$\begin{array}{l}\text{function } [x,y] = \texttt{Dot2}(u,v)\\ \quad [x,y] = \texttt{TwoProduct}(u_1, v_1)\\ \quad \text{for } i = 2:n\\ \quad\quad [h,r] = \texttt{TwoProduct}(u_i, v_i)\\ \quad\quad [x,q] = \texttt{TwoSum}(x,h)\\ \quad\quad y = \mathrm{fl}(y + (q + r))\end{array}$$

In [7, Proposition 5.5 and (5.4)] we proved the following result.

Theorem 2 *Suppose Algorithm* `Dot2` *is applied to floating point vectors $u, v \in \mathbb{F}^n$, set $s := u^T v \in \mathbb{R}$ and $S := |u^t||v|$, and suppose* $n\mathrm{eps} < 1$. *Then, also in the presence of underflow,*

$$|x + y - s| \le \gamma_n^2 S + 5n\,\mathrm{eta}, \tag{3}$$

where `eta` *denotes the smallest unnormalized positive floating point number (*$\texttt{eta} = 2^{-1074}$*).*

Using the idea in `Dot2` we can derive the following algorithm for quad precision multiplication. Input $A, B \in \mathbb{E}$ are again quad numbers with $A = [A_{hi}, A_{lo}]$, $B = [B_{hi}, B_{lo}]$.

$$\begin{array}{l}\text{function } [x,y] = \texttt{Mul}(A,B)\\ \quad [p,q] = \texttt{TwoProduct}(A_{hi}, B_{hi})\\ \quad [x,y] = \texttt{Sum2}([p, q, A_{hi} * B_{lo}, A_{lo} * B_{hi}])\end{array}$$

Note that we realize a kind of "abbreviated" multiplication by omitting the term $A_{lo} * B_{lo}$.

For quad division $X = A/B$ we use one Newton iteration applied to the function $B * X - A = 0$. Starting point is an approximation computed in double precision.

$$\begin{array}{l}\text{function } X = \texttt{Div}(A,B)\\ \quad C = A/B_{hi}\\ \quad X = C - (B * C - A)/B_{hi}\end{array}$$

The two divisions of a quad number by $B_{hi} \in \mathbb{F}$ are computed by dividing high and low order part separately. The approximation C is of double precision, and the residual $B * C - A$ is computed in quad precision. Since the Newton iteration is quadratically convergent and the correction $(B*C-A)/B_{hi}$ is computed and added separately to the double precision approximation C, the final result is of quadruple precision.

3. High precision standard functions

We briefly describe the evaluation of elementary standard functions is quadruple precision. The square root is particularly elegant to handle since the inverse function, the square of a quad number, is already available.

$$\begin{array}{l}\text{function } X = \texttt{Sqrt}(A)\\ \quad C = \sqrt{A_{hi}}\\ \quad X = (C + A/C)/2\end{array}$$

Again we apply one Newton iteration to $X^2 - A = 0$ using quad addition and division which is already available, and the double precision square root of the high order part of A as starting approximation. A little computation yields the given formula, also known as "Babylonian square root". Quadratic convergence of the Newton iteration ensures again quad accuracy of the result.

Other standard functions are a little more involved. However, we can use a table driven approach in combination with addition theorems and Taylor expansion. This is described in detail in [8]. Remarkably, an accuracy of two or three units of the last place is achieved for all common elementary standard functions.

Here we adapt the method to quad precision. Limited space allows only to sketch one function, the exponential function.

We have to calculate $\exp(x)$ for $-744.44 \le x \le 709.78$; outside this range over- or underflow is caused. Following [8] we first calculate extended precision approximations of $Y = \exp(i)$ for all integer values in that range and store

them in quad numbers Y_i, $-744 \le i \le 709$. This reduces the problem to calculate $\exp(x)$ for $0 < x < 1$. Next we calculate and store extended precision approximations $\tilde{y} = \exp(\tilde{x})$ for $\tilde{x} = i/2^{11}$ and $0 < i < 2^{11}$. This reduces the problem to calculate $y = \exp(\delta)$ for $0 < \delta < 2^{-11}$. This is done by $y = \sum_{i=0}^{8} \delta^i/i!$ with an error less than $\exp(\delta) * 2^{-99}/9! < 4.35 \cdot 10^{-36}$.

As an example of an ill-conditioned problem consider the well known question whether $\exp(\pi\sqrt{163})$ is an integer or not. Using the Matlab operator concept for a class `quad` the executable code $exp(sqrt(quad(163)) * quad(pi1, pi2))$ yields

$$262537412640768743.99999999999951\ .$$

Here $pi1$ and $pi2$ are high and low order approximations of π. However, this is no proof that $\exp(\pi\sqrt{163})$ is not an integer. In the last section we show how to *prove* $\exp(\pi\sqrt{163}) \notin \mathbb{N}$ rigorously.

4. High precision interval arithmetic and standard functions

The definition of IEEE 754 floating point standard includes directed roundings. This allows a simple and efficient way of developing high precision interval arithmetical operations and standard functions. Switching rounding mode is available in Matlab through an internal routine "system_dependent('rounding',param)". Here param=-inf or param=inf switches the rounding mode to downwards or upwards, respectively. For param=0 the rounding is set to nearest.

Denote by $\mathbb{IF}$ the set of double precision intervals, such that $X = [x_1, x_2] \in \mathbb{IF}$ with $x_1, x_2 \in \mathbb{F}$ represents $X = \{x \in \mathbb{R} : x_1 \le x \le x_2\}$. Interval operations are well known, for details see [6, 9]. A central property of interval operations is the inclusion isotonicity: For $A, B \in \mathbb{IF}$ and $C = A \circ B$ it holds

$$\forall a \in A \ \ \forall b \in B : \ a \circ b \in A \circ B\ ,$$

where $\circ \in \{+, -, \cdot, /\}$. The property applies similarly to standard functions.

We represent a quadruple precision interval A by a double precision high order part $A_{hi} \in \mathbb{F}$, and an interval $A_{lo} \in \mathbb{IF}$ as lower part. The set of quadruple precision intervals is denoted by $\mathbb{IE}$.

For the development of quadruple interval operations first note that `TwoSum` and `TwoProduct` are error-free transformations. That means, $[x, y] =$ `TwoSum`(a, b) and $[x, y] =$ `TwoProduct`(a, b) imply $x + y = a + b$ and $x + y = a \cdot b$, respectively. Therefore the for-loop in `Sum2` transforms the vector p into a new vector with identical sum. In [7] we showed that the condition number of the sum of the new vector is decreased about a factor eps.

For a given a vector $p \in \mathbb{F}^n$, the following algorithm produces a quadruple precision number $X \in \mathbb{E}$ with $X \le \sum p_i$ for rnd=-inf and $\sum p_i \le X$ for rnd=inf, respectively.

function $X =$ `Sum2rnd`(p, rnd)
  for $i = 2 : n$
    $[p_i, p_{i-1}] =$ `TwoSum`(p_i, p_{i-1})
  $X_{hi} = p_n$
  `system_dependent`$('rounding', rnd)$
  $X_{lo} = \mathrm{fl}\left(\sum_{i=1}^{n-1} p_i\right)$

Using this it is straightforward to compute interval sum and difference of quadruple intervals $A, B \in \mathbb{IE}$. Algorithm `Sum2rnd` also provides the tools for interval quadruple multiplication since the first transformation `TwoProduct` in `Mul` is error-free.

For given $A, B \in \mathbb{IE}$ the following algorithm computes an inclusion of A/B. Here m$id(A)$ denotes some value near the midpoint of A. From a mathematical point of view there is no restriction on the difference between m$id(A)$ and the exact midpoint of A.

function $X =$ `Div`(A, B)
  $C = mid(A)/B_{hi}$
  $X = C - (mid(B) * C - A)/(B_{hi} + B_{lo})$

Note that operations in the computation of X are interval computations and that $B_{hi} + B_{lo} \in \mathbb{IF}$.

For the square root we use the fact that for a given function $f : \mathbb{R} \to \mathbb{R}$, $X \in \mathbb{IR}$ and $\tilde{x} \in X$ it holds

$$\exists \hat{x} \in X \ : \ \ f(\hat{x}) = 0\texttt{quad} \Rightarrow \texttt{quad}\hat{x} \in \tilde{x} - f(\tilde{x})/f'(X)\ ,$$

where $\{f'(x) : x \in X\} \in f'(X)$ (cf. [6]). With this we see that the following algorithm calculates for given $A \in \mathbb{E}$ an inclusion of $\sqrt{A}$.

function $X =$ `ISqrt`(A)
  $\tilde{x} = \sqrt{A_{hi}}$
  $B = \sqrt{A_{hi} + A_{lo}}$
  $X = \tilde{x} - (\tilde{x}^2 - B)/(2B)$

Here the computation of B is performed using double precision interval arithmetic and double precision square root with directed rounding, a part of IEEE 754. The square root of a quad interval uses `ISqrt` and the monotonicity of the square root.

Finally, we briefly sketch the implementation of the interval exponential function. First, quad inclusions of $\exp(i) \in Y_i$ for $-744 \le i \le 709$ are calculated and stored. Next quad precision inclusions $\exp(x_i) \in y_i$ for $x_i = i/2^{11}$ and $0 < i < 2^{11}$ are calculated and stored. Finally an inclusion of $\exp(\delta)$, $0 < \delta < 2^{-11}$ is calculated using a Taylor expansion as before with interval error term.

Applying this to our previous problem to decide whether $\exp(\pi\sqrt{163})$ is an integer or not, we compute $X = exp(sqrt(quad(intval(163))) * quad(pi1, Pi2))$. Here $Pi2 \in \mathbb{IF}$ is such that $\pi \in pi1 + Pi2$. Note that the previous expression is executable code using Matlab's operator concept.

We obtain

$$
\begin{aligned}
&\mathit{intval\ quad\ X.inf} = \\
&\quad 262537412640768743.99999999999889 \\
&\mathit{intval\ quad\ X.sup} = \\
&\quad 262537412640768743.99999999999973
\end{aligned}
$$

showing that indeed $\exp(\pi\sqrt{163}) \notin \mathbb{N}$.

5. Conclusion

We showed how to implement the basic operations as well as elementary standard functions in quadruple precision, without and with error bounds. The algorithms are especially fast because branches are avoided and no special operations such as access to mantissa and/or exponent are necessary. This makes the codes highly optimizable.

The quality of a Newton step x-f(x)/f'(x) depends mainly on the accuracy of the function value f(x). Hence the precise evaluation of function values is the key to obtain high precision solutions of systems of nonlinear equations.

References

[1] T.J. Dekker. A Floating-Point Technique for Extending the Available Precision. *Numerische Mathematik*, 18:224–242, 1971.

[2] *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*, 1985.

[3] D.E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison Wesley, Reading, Massachusetts, 1969.

[4] X. Li, J. Demmel, D. Bailey, G. Henry, Y. Hida, J. Iskandar, W. Kahan, S. Kang, A. Kapur, M. Martin, B. Thompson, T. Tung, and D. Yoo. Design, Implementation and Testing of Extended and Mixed Precision BLAS. *ACM Trans. Math. Softw.*, 28(2):152–205, 2002.

[5] MATLAB User's Guide, Version 7. The MathWorks Inc., 2004.

[6] A. Neumaier. *Interval Methods for Systems of Equations*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 1990.

[7] T. Ogita, S.M. Rump, and S. Oishi. Accurate Sum and Dot Product. *SIAM Journal on Scientific Computing (SISC)*, 26(6):1955–1988, 2005.

[8] S.M. Rump. Rigorous and portable standard functions. *BIT*, 41(3):540–562, 2001.

[9] S.M. Rump. Self-validating methods. *Linear Algebra and its Applications (LAA)*, 324:3–13, 2001.