An Experiment with ACRITH

Prof. W. Kahan and Dr. Ping Tak Tang
Univ. of Calif. and Argonne National Labs.

Abstract:
We compare two paradigms for managing multi-precision floating-point arithmetic. The first is the Kulisch-Miranker paradigm as implemented in IBM's ACRITH library of High-Accuracy Arithmetic subprograms; the novelty here is the prohibition of any explicit reference to multi-precision variables while scalar products are computed as if to infinite precision. The second paradigm is a conventional one that declares some multi-precision variables into existence in a way analogous to Double-Precision declarations in contemporary languages. However, to prevent the comparison of paradigms from being confounded by irrelevant differences in their implementations, the second paradigm is implemented using only the same resources as are available for the first, namely those provided by ACRITH. And the problem chosen to be solved by both paradigms is one that must once have been thought to favor the first paradigm, since it figured in a brochure to promote ACRITH. (That brochure has since been withdrawn.) Our results illustrate why the Kulisch-Miranker paradigm is so much slower, more complex and less powerful than a conventional paradigm.

Introduction

U. Kulisch and W. Miranker and their followers advocate a novel approach to multi-precision interval arithmetic in two books (1980 and 1981) and several papers (1986). Their paradigm underlies the IBM ACRITH library of High-Accuracy Arithmetic (1983 - 1987) and two extensions, Pascal-SC and Fortran-SC (1987), of well-known higher-level programming languages. We shall begin with an unauthorised summary of their paradigm in our own terms.

Let us use ordinary letters x, y, z, ..., A, B, C, ... for floating-point variables in *working-precision*; this is IBM 370 double-precision for our experiments with ACRITH. Let us use italic letters *x*, *y*, *z*, ..., *A*, *B*, *C*, ... for *multi-precision* floating-point variables. For the present we do not distinguish interval arithmetic variables from ordinary "one-point" floating-point variables. On an IBM 370, a DEC VAX or an HP Spectrum series machine, the multi-precision variables defined within the machine's architecture are actually quadruple-precision variables. Arbitrarily high but finite precision is available to programmers in T. E Hull's Numerical Turing language (198x), in R. Brent's MP package (197x), in MACSYMA's Bigfloats (197x), etc. But the multi-precision variables in the Kulisch-Miranker scheme are restricted to sums of products of working-precision variables, which sums must be accumulated *exactly*. These multi-precision variables, called "DOT PRECISION" variables in Fortran-SC, can be added, subtracted and compared, but not otherwise combined arithmetically; and they can be rounded to working precision. They are most conveniently implemented in something called a "super-accumulator" but that's an inessential detail. Fortran-SC makes no provision for arrays of DOT PRECISION variables, each of which can consume a great deal of memory if the floating-point

exponent range is large, so a program is expected to refer to at most a few of them.

DOT PRECISION variables figure in the implementation of matrix products with only one rounding error per element. For example, to compute the product $A = B \cdot C$ of two working-precision matrices rounded just once to working precision, one DOT PRECISION variable S might be used thus:

```
for  i = 1 to I  do
for  j = 1 to J  do
  {  S := 0.0 ;
     for  m = 1 to M  do  S := S + b_i,m • c_m,j  exactly ;
     a_i,j := S  rounded to working precision  } .
```

The Kulisch-Miranker paradigm takes such an implementation for granted, so every product of two working-precision matrices can be presumed to have been computed with just one rounding error per element. An important special case is a sum $s := \sum_1^N x_n$, which may be regarded as a matrix product of a row $(x_1 \ x_2 \ \ldots \ x_N)$ by a column of 1's evaluated with one final rounding error.

How can DOT PRECISION variables be used to compute anything else than one matrix product accurately? The crucial observation is that any polynomial can be evaluated to within less than one ulp (unit in the last place) of working precision by using *iterative refinement*, a technique to be discussed in a moment. A rational function can be transformed into a ratio of two polynomials, so it can be calculated correctly to within a few ulps. And an algebraic function is the solution of a polynomial equation that can be solved by Newton's iteration to within an ulp provided multiple roots do not get in the way. Therefore, any algebraic function can be computed to within a few ulps, except too close to its singularities; and, provided neither exponent over/underflow nor memory exhaustion intervenes, that can be done without any other kind of multi-precision arithmetic than once-rounded matrix multiplication. But there is a catch. ...

Iterative Refinement to multiply several matrices:
In its place, there is nothing wrong with iterative refinement. But the Kulisch-Miranker paradigm uses it almost everywhere, and that turns out to be a bad idea. Here is how it goes:

Suppose that we wish to compute an expression z that turns out to be a polynomial of total degree N in working-precision data a, b, c, ... For example, $z = (a + b \cdot c) \cdot (d + f \cdot v)$ is of total degree $N = 4$. Then there exists a way to express z as a matrix product $z = A_1 \cdot A_2 \cdot \ldots \cdot A_N$ in which all elements of each matrix A_n are simple integer constants or else data items. For our example, $z = (a \ b) \cdot \begin{pmatrix} 1 \\ c \end{pmatrix} \cdot (d \ f) \cdot \begin{pmatrix} v \\ 1 \end{pmatrix}$. Many other matrix products can serve equally well except for the amount of work entailed; the trick is to find a matrix product representation of the polynomial that minimizes the work. Aside from that consideration, we see now that the computation of any polynomial is a special case of the problem of computing a product of several matrices.

In the obvious recurrence to compute the product of N matrices

we observe that each intermediate product appears only linearly:
$x_1 = A_1$; $x_2 = x_1 \cdot A_2$; $x_3 = x_2 \cdot A_3$; ... ; $z = x_N = x_{N-1} \cdot A_N$.
Therefore this recurrence can be interpreted as a way to solve a linear system of equations $X \cdot A = B$ where

$$X = (x_1 \quad x_2 \quad x_3 \quad \dots \quad x_N) ,$$
$$B = (A_1 \quad 0 \quad 0 \quad \dots \quad 0) , \quad \text{and}$$

$$A = \begin{pmatrix} 1 & A_2 & 0 & \dots & 0 \\ -1 & A_3 & 0 & \dots & 0 \\ & -1 & A_4 & \dots & \dots \\ & & \dots & \dots & 0 \\ & & & -1 & A_N \\ & & & & -1 \end{pmatrix} .$$

For our example, $B = (a \quad b \quad 0 \quad 0 \quad 0 \quad 0)$ and

$$A = \begin{pmatrix} 1 & 0 & 1 & & & \\ & 1 & c & & & \\ & & -1 & d & f & \\ & & & -1 & 0 & 1 \\ & & & & -1 & v \\ & & & & & -1 \end{pmatrix} , \quad \text{so}$$

$$X = (\underset{x_1}{a} \quad \underset{x_2}{b} \quad \underset{x_3}{a+bc} \quad \underset{x_4}{(a+bc)d} \quad (a+bc)f \quad z) .$$

Since A is an upper triangular matrix, the equation $X \cdot A = B$ can be solved for X in the obvious way; but rounding every product $x_{n-1} \cdot A_n$ once to working precision hardly ever yields a final x_N computed accurately to within an ulp. Let X_0 be the estimated X computed that way and then compute $B_0 = (B - X_0 \cdot A)$ rounded once. The equation $X_1 \cdot A = B_0$ can be solved for X_1 in the same way, and then $B_1 = (B - X_0 \cdot A - X_1 \cdot A)$ can be computed rounded once if needed. In general, until the once-rounded value of $(X_0 + X_1 + X_2 + \dots X_k)$ converges to X within an ulp, as can be ascertained with the aid of interval arithmetic in a way to be discussed later, we compute repeatedly the residual

$$B_k = (B - X_0 \cdot A - X_1 \cdot A - X_2 \cdot A - \dots - X_k \cdot A)$$

rounded once to working precision, and then compute another term X_{k+1} by "solving" $X_{k+1} \cdot A = B_k$. This is *iterative refinement*, also called "the method of deferred corrections." When used to compute polynomials z in this way it always converges in at most a finite number k of steps unless over/underflow intervenes.

What is wrong with iterative refinement? It is too slow for three reasons. First, because we cannot save arrays of DOT PRECISION variables, the residual B_k must be computed from the formula above at a cost of $k+1$ multiplications by the matrix A instead of from the algebraically equivalent recurrence

$R_{-1} = B$; $R_k = R_{k-1} - X_k \cdot A$ exactly ; $B_k = R_k$ rounded

at a cost of just one multiplication per step. Consequently X_k costs $(k+1)(k+2)/2$ multiplications by A instead of just $2k+1$.

Second, iterative refinement creeps up on the precision required to compute X accurately enough instead of jumping to it directly as does the more conventional paradigm to be described presently. When $(X_0 + X_1 + X_2 + \dots + X_k)$ is accurate enough to round once to X within an ulp, it occupies at least as much memory as a $(k+1)$-times working-precision computation would occupy, and cost

at least about $(k+1)/2$ as much time (as we shall see), yet must be less accurate because each deferred correction term X_k adds less than one working-precision word of accuracy to the sum.

Third, the requirement, that DOT PRECISION sums be accumulated *exactly* instead of merely to somewhat more than the $(k+1)$-times working-precision that is actually needed, must retard progress to some extent. This retardation is difficult to gauge without knowledge of implementation details that may well vary from one machine to another, so we shall not try to take it into account. Instead, we shall implement multi-precision arithmetic on top of DOT PRECISION, so that each of our multi-precision operations will take at least as long as the corresponding DOT-PRECISION operation would have taken if that operation were available. In doing so, we bias our comparison in favor of the Kulisch-Miranker paradigm, which is the conservative thing to do.

The worst aspect of the Kulisch-Miranker paradigm is not that it runs slower than ordinary multi-precision arithmetic, but that it is more complicated. Compare the foregoing algorithm to multiply several matrices with the one that follows.

Multi-Word Precision multiplication of several matrices.
Recalling that any polynomial of total degree N can be expressed as a product of N matrices, we turn now to the computation of such a product $Z = A_1 \cdot A_2 \cdot \ldots \cdot A_N$ with the aid of several k-times working-precision variables $S_2, S_3, \ldots, S_N$, where k is a parameter that can be varied, within limits, at run-time. The only operations that will be performed upon these k-tuple precision variables here will be addition, multiplication by a working-precision variable, and rounding to working-precision, each of which consumes time proportional to k at worst. At the start we set $k = 1$ or 2; later the computation will be repeated with as large a value of k as appears necessary. The algorithm that multiplies N matrices is a natural generalization of the case $N = 2$ described above, but now we have so many subscripts that the exigencies of typography force a Fortran-like notation for them. We denote the element of A_n in its i^{th} row and j^{th} column by $A_n[i,j]$ for $1 \le i \le I_n$ and $1 \le j \le I_{n+1}$. Let us first review the ordinary algorithm to compute a product of two matrices, $Z = A_1 \cdot A_2$, in that notation:

```
for  i = 1 to I1  do
for  j = 1 to I3  do
    {  S2 := 0 ;
       for  m = 1 to I2  do  S2 := S2 + A1[i,m]•A2[m,j] ;
       Z[i,j] := S2  }.      ...   Z = A1•A2 .
```

Here the product $A_1[i,m] \cdot A_2[m,j]$ must be evaluated exactly as a k-tuple precision number when $k > 1$, which costs no more work when $k > 2$ than when $k = 2$, before being added to S_2. For a product $Z = A_1 \cdot A_2 \cdot A_3$ of three matrices, a product $S_2 \cdot A_3[m,j]$ must be rounded to k-tuple precision before it is added to another k-tuple precision number S_3 in the following program:

```
  for  i = 1 to I1  do
  for  j = 1 to I4  do
    { S3 := 0 ;
      for m = 1 to I3  do
        {  S2 := 0 ;
           for  l = 1 to I2  do  S2:= S2 + A1[i,l]•A2[l,m] ;
           S3 := S3 + S2•A3[m,j] } ;
       Z[i,j] := S3  }.      ...   Z = A1•A2•A3 .
```

A product of N matrices A_n , with *variable* $N \geq 2$, requires a *recursive* program like this:

```
  procedure  MatxProd( N, Ac3[,], Ic3, Z[,], Sc3, k ) ;
   integer  N, IN+1, k ;          ...  Ic3  is an integer array.
   real  Z[,], AN[,] ;            ...  An  must be  In by In+1 .
   real extended*k  SN ;          ...  Sn  is  k-tuple precision.
   { integer  i, j ;

     recursive sub-procedure  WideAdd( n, j ) ;
      integer  n, j ;               ...  n ≥ 2 .
      { integer  m ;
        Sn := 0 ;
        if  n = 2  then             ...      ( k-tuple precision )
             { for  m = 1 to I2  do  S2 := S2 + A1[i,m]•A2[m,j] }
                  else
             { for  m = 1 to In  do
                                  { call  WideAdd( n-1, m ) ;
                                    Sn := Sn + Sn-1•An[m,j] } } ;
        return ;                    ...  ( k-tuple precision )
        end  WideAdd ;

     if  n < 2  then
      { print "ERROR: MatxProd( N, ...) expects  1 < N = ", N ;
        STOP } ;
                 else
      { for  i = 1 to I1    do
        for  j = 1 to IN+1  do
          { call  WideAdd( N, j ) ;
            Z[i,j] := SN  rounded to working ("real") precision };
        return } ;
        end  MatxProd .    ...   Z = A1•A2•...•AN
```

Of course, if N is a constant known in advance, the recursive call upon WideAdd can be expanded in-line so that recursion may be removed; the reader should try this for N = 3 or N = 4 to be reassured first that the program is correct and, second, that otherwise recursion is practically unavoidable. (No wonder that programs like this are not written in Fortran !)

How does the speed of MatxProd compare with that of iterative refinement as used for matrix multiplication above? That depends upon the time taken to multiply (working prec.)•(k-tuple prec.) and to add (k-tuple prec.)+(k-tuple prec.) numbers. Both take times proportional to k and rather longer than the time for one exact (working prec.)•(working prec.) + (DOT PRECISION) operation in the iterative refinement process. A conservative assumption is that each k-tuple precision multiply-add takes no more than k times as long as this exact DOT PRECISION multiply-add. With

this assumption, the recursive program takes less time than did the k^{th} step of iterative refinement. Invoking MatxProd first with $k = 2$, at a cost roughly equal to the first step of iterative refinement, produces an estimate of Z whose accuracy can be assessed either by means of Interval Arithmetic or by a conventional running error-analysis; details will be supplied later. Iterative refinement requires the same kind of assessment and can get it the same way at the same cost. Then, if the accuracy is deemed inadequate, MatxProd will have to be rerun with a larger value of k. That value need only be large enough that reducing the error assessment by a factor ε^{k-2}, where ε is the relative error bound for each working-precision arithmetic operation, would render the error negligible to working precision. This k can be no larger than the last k needed for iterative refinement, so the rerun of MatxProd will consume no more time than the last step of iterative refinement. That is why iterative refinement take about (k+1)/2 times as long as two calls upon MatxProd to deliver a product Z within about one ulp of working precision.

These estimates of comparative speed are consistent with results reported by Kahan and LeBlanc (1985), who found that ACRITH's version of iterative refinement took more than ten times as long than what they called "a renegade algorithm" (an algorithm like the one described above using DOT PRECISION variables to simulate k-tuple precision arithmetic) to compute the product of three 4 by 4 matrices. Now we understand why ACRITH is so slow; but it is not so slow as to be unusable. Normally k does not get very big, rarely bigger than 2, so speed can be an important issue only on rare occasions when very high precision is necessary. If this were the only thing wrong with the Kulisch-Miranker paradigm, it would not be worth complaining about.

```
**************************************************************
*  The crucial defect in the Kulisch-Miranker paradigm is    *
*  that programs like MatxProd can evaluate more diverse     *
*  products than iteative refinement can. Polynomials exist  *
*  whose values, and all the intermediate values generated   *
*  during their evaluation, are entirely innocuous; and      *
*  programs like MatxProd can compute them, but iterative    *
*  refinement cannot. And some of these polynomials have     *
*  appeared among the examples used to promote ACRITH and the *
*  Kulisch-Miranker paradigm, but they were not evaluated at  *
*  arguments that would have revealed this defect.           *
**************************************************************
```

We shall exhibit some of those polynomials and explain their behavior. TO BE CONTINUED