18.9 Extra-Precision Accumulation

A fundamental operation in numerical linear algebra, first mentioned in §6.3, is finding the **dot product** of two vectors $\mathbf{x}$ and $\mathbf{y}$ as the following sum.

$$\mathbf{x}^{\top}\mathbf{y} = \sum_{j=1}^{n} x_j y_j.$$

This calculation is likely to be imprecise because of rounding error in the multiplications and cancellation error when small terms are added into a large sum, as discussed in §4.3. I mentioned there that cancellation error can be reduced by adding up the terms in ascending order of absolute value, but that is seldom done in finding the dot product because precomputing and sorting the products $x_j y_j$ uses significant extra memory and CPU time. Nonetheless we often want a precise answer, so it is standard practice to instead accumulate the sum at **extra precision**. For example, if the basic calculation uses `REAL*4` numbers the dot product might be coded using `REAL*8` arithmetic like this.

```
      REAL*4 X(100),Y(100),DOT
      REAL*8 Z
      :
      Z=0.D0
      DO 1 J=1,100
           Z=Z+DBLE(X(J))*DBLE(Y(J))
    1 CONTINUE
      DOT=SNGL(Z)
      :
```

Here the `DBLE` function (see §4.4) is used to cast `X(J)` and `Y(J)` to `REAL*8` for the multiplication, and `SNGL` is used to convert the result `Z` back to `REAL*4`. If your compiler supports the `REAL*16` data type, you can modify this code to compute accurate `REAL*8` dot products. But what if your compiler does *not* recognize `REAL*16`, or it does but the basic calculation already uses `REAL*16` and you want more precision than that? There is in fact a clever way (see Stokes, and §4.3.3 of Knuth Volume 2) to perform the dot product calculation at extra precision with variables of the same precision as those used to store the vectors, and with only a small penalty in memory and processor time.

Multiplying two n-bit binary fractions a and b yields a product ab that is $2n$ bits long, as in this example with $n = 4$.

```
          • 1 1 1 0 = a
        × • 1 1 0 1 = b
          ---------
            1 1 1 0
          0 0 0 0
        1 1 1 0
      1 1 1 0
    ---------------
    • 1 0 1 1 0 1 1 0 = ab
```

To store this result as a 4-bit binary fraction we must discard the least-significant 4 of its fraction bits, or *half* of the bits that make up the answer! These bits are of course much less important than the ones we keep, but neglecting them does introduce some error. The right answer is $.10110110_2 = \frac{1}{2} + \frac{1}{8} + \frac{1}{16} + \frac{1}{64} + \frac{1}{128} = 0.7109375_{10}$ but the result we keep is $.1011_2 = 0.6875_{10}$.

Instead suppose we **split** a into two parts so that $a = a_h + a_t$, where a_h is the value of the high or most-significant $n/2$ bit positions of a and a_t is the value of the trailing or least-significant $n/2$ bit positions. Then, if a_h and a_t are stored as floating-point binary fractions having n significand bits, the rightmost $n/2$ bits in each of them will be zero. Splitting b will yield parts b_h and b_t that similarly have zeros in their $n/2$ least-significant bit positions. Then we can find the product as

$$ab = (a_h + a_t)(b_h + b_t) = a_h b_h + a_h b_t + a_t b_h + a_t b_t$$

where each partial product is *exactly* represented by a floating-point binary fraction of n bits and can therefore be stored without any loss of precision. For our $n = 4$ example this is how the process works.

$$\begin{aligned}
a &= .1110 = .1100 \times 2^0 + .1000 \times 2^{-2} = a_h + a_t \\
b &= .1101 = .1100 \times 2^0 + .0100 \times 2^{-2} = b_h + b_t \\
a_h b_h &= (.1100 \times 2^0\ \ ) \times (.1100 \times 2^0\ \ ) = .1001 \times 2^0 \\
a_h b_t &= (.1100 \times 2^0\ \ ) \times (.0100 \times 2^{-2}) = .0011 \times 2^{-2} \\
a_t b_h &= (.1000 \times 2^{-2}) \times (.1100 \times 2^0\ \ ) = .0110 \times 2^{-2} \\
a_t b_t &= (.1000 \times 2^{-2}) \times (.0100 \times 2^{-2}) = .0010 \times 2^{-4}
\end{aligned}$$

Each of the parts has $n/2 = 2$ trailing zeros, and each partial product just fits in $n = 4$ bits. If we align binary points and add partial products we get the same answer as before.

$$\begin{array}{l}
\bullet 10010000 = a_h b_h \\
\bullet 00001100 = a_h b_t \\
\bullet 00011000 = a_t b_h \\
\bullet 00000010 = a_t b_t \\
\hline
\bullet 10110110 = ab
\end{array}$$

To avoid losing the least-significant half of this result, we could accumulate the sum of the partial products into a two-element vector of 4-bit floating-point binary fractions, ending up with $ab = [(.1011 \times 2^0), (.0110 \times 2^{-4})]$. Once a whole dot product has been accumulated, the less-significant parts of all the partial products will have added up instead of being lost through cancellation, and we can obtain an accurate n-bit answer by adding the two n-bit vector elements that we used to store the $2n$-bit sum.

The `MPYACC` subroutine listed on the next page uses the splitting idea to perform a single multiplication of the scalar `X` times the scalar `Y`, calling `ADDACC` to add each partial product to the two-element accumulator `XYSUM`. Unlike a and b in the discussion above, `X` and `Y` are $\boxed{21}$ `REAL*8` variables. According to §4.2 they have a sign bit and 11 exponent bits preceding an implied "`1.`" and 52 bits of binary fraction, so in splitting them it is necessary to preserve the sign and exponent bits. To split `X` we begin $\boxed{35}$ by copying it into `XH`, which is $\boxed{25\text{-}26}$ overlaid by the two-element `INTEGER*4` vector `IXH`. On a little-endian processor the least-significant word of `X` comes first in memory (see §4.8) so another name for it is `IXH(1)`. This fullword we bitwise-AND (see §4.6.3) with `HMASK` $\boxed{37}$ which is initialized $\boxed{27}$ at compile time to the bit pattern `11111100000000000000000000000000`. The resulting value of `XH` is thus `X` with its least-significant 26 (= $n/2$ in the discussion above) bits set to zero. We want `XH` and `XT` to add up to `X`, so $\boxed{38}$ `XT` is just `X` minus the `XH` we found. The same process is used $\boxed{39\text{-}42}$ to split `Y` into `YH` and `YT`. The parts `XH`, `XT`, `YH`, and `YT`, are `REAL*8` so they have 52 fraction bits, but of these the trailing 26 are zero. Finally the code $\boxed{45\text{-}52}$ computes the four 52-bit partial products (in order from smallest to largest) and adds each to the extra-precision accumulator.

```
      SUBROUTINE MPYACC(X,Y, XYSUM )
C     This routine accumulates XYSUM=XYSUM+X*Y at extra precision.
C
C     variable  meaning
C     --------  -------
C     ADDACC    routine adds to an extra-precision accumulator
C     HMASK     deletes the 26 least-significant fraction bits
C     IAND      Fortran function for bitwise logical AND
C     IXH       XH as 2 fullwords
C     IYH       YH as 2 fullwords
C     P         a partial product
C     X         first number in product
C     XH        split of X containing its high 26 fraction bits
C     XT        split of X containing value of trailing 26 bits
C     XYSUM     extra-precision accumulator
C     Y         second number in product
C     YH        split of Y containing its high 26 fraction bits
C     YT        split of Y containing value of trailing 26 bits
C
C     formal parameters
      REAL*8 X,Y,XYSUM(2)
C
C     prepare to split X and Y
      REAL*8 XH,XT,YH,YT
      INTEGER*4 IXH(2),IYH(2)
      EQUIVALENCE(XH,IXH),(YH,IYH)
      INTEGER*4 HMASK/Z'FC000000'/
C
C     prepare to compute the partial products
      REAL*8 P
C
C -----------------------------------------------------------------
C
C     split X and Y into parts having 26 trailing fraction bits zero
      XH=X
C     this assumes the processor is little-endian
      IXH(1)=IAND(IXH(1),HMASK)
      XT=X-XH
      YH=Y
C     this assumes the processor is little-endian
      IYH(1)=IAND(IYH(1),HMASK)
      YT=Y-YH
C
C     add the 52-fraction-bit exact partial products to accumulator
      P=XT*YT
      CALL ADDACC(P,XYSUM)
      P=XT*YH
      CALL ADDACC(P,XYSUM)
      P=XH*YT
      CALL ADDACC(P,XYSUM)
      P=XH*YH
      CALL ADDACC(P,XYSUM)
      RETURN
      END
```

The additions are accomplished by the ADDACC subroutine, which is listed on the next page. ADDACC begins $\boxed{\texttt{24-30}}$ by putting the larger of P and XYSUM(1) in U and the smaller in V. This is to minimize cancellation error in the calculation $\boxed{\texttt{36}}$ of U-Z (if U is close to Z=U+V then little or no shifting will be needed to align the binary points in finding U-Z). Then $\boxed{\texttt{33}}$ we find Z=U+V. Here some of the less-significant fraction bits of V are probably lost because V must be shifted to align its binary point with that of U. How much error does that introduce? The difference U-Z should be exactly -V, but because of cancellation it will differ from -V by the error we seek. This is calculated $\boxed{\texttt{36}}$ as ZZ. To that we add $\boxed{\texttt{39}}$ the current contents of the least-significant doubleword of the accumulator. If the least-significant doubleword has grown big enough to be noticed if we added it to the most-significant doubleword, we want to move that much of it there. So the most-significant doubleword of the accumulator then becomes $\boxed{\texttt{42}}$ the imprecise sum plus the correction to the sum plus the least significant doubleword of the accumulator. Finally $\boxed{\texttt{45}}$ we replace the least-significant doubleword of the accumulator with the (small) amount that is necessary to make XYSUM(1)+XYSUM(2) equal to the corrected sum Z+ZZ. The complicated process just described has the effect of adding P to XYSUM at $2 \times 52 = 104$ bits of precision, which is almost the 112 bits of precision we would get if we were able to use REAL*16 arithmetic.

```
      SUBROUTINE ADDACC(P, XYSUM )
C     This routine adds P to the extra-precision accumulator XYSUM.
C     It must be compiled with optimization turned off.
C
C     variable  meaning
C     --------  -------
C     DABS      Fortran function returns |REAL*8|
C     P         quantity to be added to the accumulator
C     U         the larger in absolute value of P and XYSUM
C     V         the smaller in absolute value of P and XYSUM
C     XYSUM     the accumulator
C     Z         most significant part of sum
C     ZZ        least significant part of sum
C
C     formal parameters
      REAL*8 P,XYSUM(2)
C
C     local variables
      REAL*8 U,V,Z,ZZ
C
C -----------------------------------------------------------------
C
C     put the larger quantity in U and the smaller in V
      IF(DABS(XYSUM(1)) .LT. DABS(P)) THEN
         U=P
         V=XYSUM(1)
      ELSE
         U=XYSUM(1)
         V=P
      ENDIF
C
C     find the sum, imprecisely
      Z=U+V
C
C     compute the error that was made by rounding U+V to REAL*8
      ZZ=(U-Z)+V
C
C     add to it the least significant part of the accumulator
      ZZ=ZZ+XYSUM(2)
C
C     that might be enough to increase the most significant part
      XYSUM(1)=Z+ZZ
C
C     make the least significant part of accumulator what is left
      XYSUM(2)=(Z-XYSUM(1))+ZZ
C
      RETURN
      END
```

The DDOTQ function listed on the next page uses MPYACC to compute a dot product using extra-precision accumulation. After doing some sanity-checking $\boxed{\texttt{23-24}}$ it initializes the accumulator XYSUM to zeros $\boxed{\texttt{27-28}}$. Instead of the multiply-and-add loop we had before we now have $\boxed{\texttt{29-31}}$ a loop of calls to MPYACC. On each invocation that routine computes X(J) $\times$ Y(J) and adds it to the accumulator as described above. When the loop is finished we find the dot product $\boxed{\texttt{34}}$ by adding together the most- and least-significant doublewords of the accumulator.

The program below compares DDOTQ to DDOT for finding a troublesome dot product.

```
      REAL*8 X(101),Y(101),DDOT,ANS,DDOTQ,ANSQ
      X(1)=1.D+08
      Y(1)=1.D+08
      DO 1 J=2,101
           X(J)=DFLOAT(J-1)
           Y(J)=1.D0/DFLOAT(J-1)
    1 CONTINUE
      ANS=DDOT(X,Y,101)
      ANSQ=DDOTQ(X,Y,101)
      WRITE(6,901) ANS,ANSQ
  901 FORMAT('DDOT  finds ',1PD23.16/
     ;       'DDOTQ finds ',1PD23.16)
      STOP
      END
```

The program manufactures the following problem.

$$\mathbf{x} = [10^8, 1, 2, 3, \ldots, 100]$$
$$\mathbf{y} = [10^8, 1, \tfrac{1}{2}, \tfrac{1}{3}, \ldots, \tfrac{1}{100}]$$
$$\mathbf{x}^\top\mathbf{y} = 10^{16} + (1 \times 1) + (2 \times \tfrac{1}{2}) + (3 \times \tfrac{1}{3}) + \cdots + (100 \times \tfrac{1}{100}) = 10000000000000100$$

When the program is compiled with gfortran and run, it produces the following output. The product of the first two terms, 10^{16}, is big enough so that the subsequent terms contribute nothing to the sum when DDOT does the calculation using REAL*8 arithmetic. However, when DDOTQ does the calculation using extra-precision accumulation the correct result is obtained.

```
unix[1] a.out
DDOT  finds  1.0000000000000000D+16
DDOTQ finds  1.0000000000000100D+16
unix[2]
```

This chapter has introduced two-part values, which can be used to perform fixed-point arithmetic with numbers too big to store in an INTEGER*4, and extra-precision accumulation for computing floating-point dot products more precisely than we can by simply doing REAL*8 arithmetic. It is also possible to use Classical FORTRAN for integer calculations of *arbitrary* precision, as described in §20.6 of *Numerical Recipes* for example, and for floating-point calculations of *arbitrary* precision by invoking Brent's multiple precision package.

```
      FUNCTION DDOTQ(X,Y,N)
C     This routine computes the dot product of X with Y,
C     using extra-precision accumulation.
C
C     variable  meaning
C     --------  -------
C     J         index on the elements of X and Y
C     MPYACC    routine does extra-precision multiply and accumulate
C     N         number of elements in X and Y
C     X         one of the vectors in the dot product
C     XYSUM     extra-precision result
C     Y         the other vector in the dot product
C
C     formal parameters
      REAL*8 DDOTQ,X(N),Y(N)
C
C     local variable
      REAL*8 XYSUM(2)
C
C ------------------------------------------------------------------
C
C     check for a sensible value of N
      DDOTQ=0.D0
      IF(N.LE.0) RETURN
C
C     accumulate the product at extended precision
      XYSUM(1)=0.D0
      XYSUM(2)=0.D0
      DO 1 J=1,N
           CALL MPYACC(X(J),Y(J), XYSUM )
    1 CONTINUE
C
C     return a double-precision answer
      DDOTQ=XYSUM(1)+XYSUM(2)
      RETURN
      END
```

Reference

The paper mentioned on page 1 is **Stokes, H. H.**, "The sensitivity of econometric results to alternative implementations of least squares," *Journal of Economic and Social Measurement* 30 (2005) 9-38. In the source code of Stokes' B34S program, this approach to implementing the extra precision accumulation idea is attributed to "1980 IMSL code that is no longer supported."

License